Abstract

There are two different approaches for estimation of structure and/or motion of objects in the computer vision community today. One is the feature correspondence method, and the other is the optical flow method [1]. There are many difficulties and limitations encountered with the feature correspondence method, while the optical flow method is more feasible, but requires a substantial amount of extra calculations if the optical flow is to be computed as an intermediate step.

Direct methods have been developed [2-4], that use the optical flow approach, but avoid computing the full optical flow field as an intermediate step for recovering structure and motion. The unified optical flow field theory was recently established in [5]. It is an extension of the optical flow (UOFF) [1] to stereo imagery. Based on the UOFF, a direct method is developed to reconstruct an Alpha shape surface structure characterized by an third degree polynomial equation, and a Sphere surface characterized by a second degree polynomial [6]. This thesis work uses the methods developed in [5,6], to reconstruct the third degree polynomial describing a surface.

The main difference from the simulation results obtained in [6], is that in this case, one of the two surfaces tested is a third order, unbounded surface, and that the image gradients are computed directly from the image data, with no prior knowledge of the surface gray function distribution. Another important difference is that the gray levels of the surface are quantized in this work; i.e., the computations are done using integer image data, not the continuous gray levels as in [6]. These differences contribute to proving that the UOFF technique can be used in a practical manner, and with good results.

Further discussions of the contributions of this work are included in the last chapter.

2) # DIRECT RECOVERING OF SURFACE STRUCTURE CHARACTERIZED BY AN Nth DEGREE POLYNOMIAL EQUATION USING THE UOFF APPROACH

by

1) Mazen A. Salhi

A Thesis

Submitted to the Faculty of the Graduate Division of the
New Jersey Institute of Technology

in Partial Fulfillment of the Requirements for the Degree of

Master of Science in Electrical Engineering

Department of Electrical and Computer Engineering

October 1991

Title: Direct Recovering of Surface Structure Characterized

By an Nth Order Polynomial Using the UOFF Approach

Name: Mazen A. Salhi

Thesis and Defence Approved

Dr. Yun Q. Shi Advisor

Assistant Professor

Dr. Chang-Qimg Shu

Research Associate

Dr. John Carpinelli

Assistant Professor

Vita

Name:	Mazen Abdul-Rahim Salhi
Degree to be conferred:	M.S. Electrical Engineering, October, 1991.
Secondary Education:	Jeddah Secondary School, 1984
Undergradute College:	King Fahd Universitiy of Petroleum and Minerals
Dates:	September 1984 to August 1989
Degree:	B.S. Electrical Engineering
Date of Degree:	August 1989
Graduate School:	New Jersey Institute of Technology
Dates:	September 1989 to May 1991
Degree:	M.S. Electrical Engineering
Date of Degree	October 1991

Aknowledgement

I would like to extend my gratitude to the many people who have helped and contributed significantly, in the bringing about of this thesis. In particular, I would like to thank Dr. Yun Q. Shi, my thesis advisor, for his ever present support and follow up, both technically, and interpersonally. I am also thankful to Dr. Chang-Qimg Shu, and Dr. John Carpinelli for their valuable advice and assistance, in completing this thesis.

I also extend my deep appreciation to my parents, whose care and encouragement have always been a major inspiration for me.

Contents

List of Tables

List of Figures

Chapter 1

INTRODUCTION

Estimation of structure and motion from image sequences has become a major research field in the computer vision community over the last ten years. There are basically two different approaches to recovering the structure of an object, or the relative motion between the object(s) and the camera(s): the optical flow field approach, and the feature correspondence approach.

Feature-based methods require the solution of the *feature detection* and the *correspondence* problems, which have proven to be computationally and conceptually difficult to solve. These methods are also highly sensitive to noise since information from only a small portion of the image is used. Furthermore, identifying features involves determining gray-level *corner* points and studying a correlation problem between the various segments of consequent images; while for images of smooth objects, it is difficult to identify good features and corners.

Flow-based methods have been proposed that use the distribution of apparent velocities of movement of image brightness patterns, the so called *optical flow*, over a large portion of the image, to recover depth and/or camera motion [1]. These methods achieve more robustness at the expense of more computation to determine an optical flow field that is accurate enough for analysis. To compute the full optical flow field however, one needs additional constraints such as the heuristic assumption that the flow field is locally smooth [3]. This in many cases leads to an estimated optical flow field that is not the same as the true motion field.

To avoid these problems, direct methods [2-4] have been proposed that are more robust since information about the whole image is employed, and which require less computation since readily computable data (image brightness gradients in temporal and spatial domains) are used directly to extract depth and motion information.

Recently, a new approach to motion analysis from a sequence of stereo images has been developed, in which the optical flow determined by Horn and Schunck [1] is extended to include a spatial image sequence [5]. This results in a unified optical flow field combining both the spatial and the temporal cases. A set of equations is established to characterize the UOFF. Another set of equations is derived to recover the structure.

There is no need for feature correspondence in this approach. The recovered 3-D structure is for a whole continuous field. It is therefore clearly, an optical flow approach. There are two major aspects of the UOFF concept: The first is that the brightness function of an image is considered not only as a function of time, but also a function of the various sensors' spatial positions. And the second is that the brightness invariance equation is recognized not only for the time variation but also for the space variation so that both the time and space domains are included in a single, *unified* brightness invariance equation. It is noted that the optical flow for a temporal image sequence discussed in [1] is a special case within the framework of the UOFF.

Based on the UOFF, a direct method was developed to reconstruct a surface structure characterized by an Nth degree polynomial [6]. The new method does not require the computation of the UOFF quantities explicitly as an intermediate step.

In this thesis, the new concept of the UOFF is studied, and its *spatial* case is applied in reconstructing an Nth degree polynomial describing a surface structure. Simulation images for a 2-nd and a 3-rd order surfaces viewed from two cameras are used, and the structure is recovered by optimizing a performance function which derives from the solution of the UOFF.

1.1 Preliminaries

1.1.1 Imaging Space

Consider a sensor located in a specific position in 3-D world space, that keeps generating images about the scene . As time goes by, the sensor at this particular position in 3-D space, forms a sequence of images. The set of these images can be represented with brightness function $g(x, y, t)$, where x and y are coordinates on the image plane. This is the basic outline about brightness function $g(x, y, t)$ treated by Horn and Schunck [1].

A different sequence of images can be formed as follows. If at a specific moment in time, there are infinitely many sensors in the Imaging space to view the object from all possible different positions, then we cannot use the previous brightness function $g(x, y, t)$ to describe the gray levels of the image plane. Combining the two factors of time and space, we obtain yet another, much larger, set of images. To describe the brightness of this new set, we need a more general brightness function

$$g(x, y, t, \vec{s}), \tag{1.1}$$

where $\vec{s}$ indicates the sensor's position in 3-D world space, i.e., the coordinates of the sensor center and the orientation of the optical axis of the sensor . As mentioned previously $\vec{s}$ is a 5-D vector. That is

$$\vec{s} = (\tilde{x}, \tilde{y}, \tilde{z}, \beta, \gamma) \tag{1.2}$$

where $\tilde{x}$, $\tilde{y}$ and $\tilde{z}$ represent the coordinate of the optical center of the sensor in 3-D world space; β and γ represent the orientation of the optical axis of the sensor in 3-D world space.

More specifically, each sensor in 3-D world space may be considered associated with a 3-D Cartesian coordinate system such that its center is located on the origin and its optical axis is aligned with the OZ axis. We choose in 3-D world space a 3-D Cartesian coordinate system as the reference coordinate system. Hence, a sensor with its associated Cartesian coordinate system coincident with the reference coordinate system, has its position in 3-D world space denoted by $\vec{s} = (0,0,0,0,0)$. An arbitrary sensor position denoted by $\vec{s} = (\tilde{x}, \tilde{y}, \tilde{z}, \beta, \gamma)$ can be described as follows. The sensor's associated Cartesian coordinate system has been shifted first from the reference coordinate system in 3-D world space with its origin settled at $(\tilde{x}, \tilde{y}, \tilde{z})$ in the reference coordinate system and then has been rotated with the rotation angles β, about its OY axis, and, γ, about its OX axis, being the Euler angles: pan and tilt, respectively.

1.1.2 Setup

The geometry of the setup is that of a typical stereo system (Fig. 3.5). There is a world Cartesian coordinate (X, Y, Z), and two identical cameras positioned as shown. In 3-D space, a camera can be translated with three degrees of freedom, and rotated with two degrees of freedom. The rotation of a camera around its optical axis is not considered since no change in the image information will result. The optical axis of the *left* camera coincides with the Z axis of the world coordinate system, and the center of the left image plane is located at $(0,0,1)$ exactly.

The surface structure has its own coordinate system. The origin of this system is O', which differs from O by a sole translation of distance D, in the positive direction of Z. This coordinate system is used to make the description of the surface structure easier later and to avoid numerical problems. The value of D is much larger than 1. For the *right* camera, the image plane is rotated with β degrees, and translated in the X, and Z directions, maintaining a constant distance, D, between the center of the right image plane and the origin of the surface structure.

1.2 Perspective Projection

A world point P in 3-D space is projected onto the image plane according to perspective projection, and is located on the image plane at $x_P$and y_P, where,

$$x_P = \frac{X}{Z} \tag{1.3}$$

$$y_P = \frac{Y}{Z} \tag{1.4}$$

From our brightness function discussed above, x_P and y_P are also dependent on t and $\vec{s}$. That is, the coordinates of the pixel can be denoted by $x_P = x_P(t, \vec{s})$ and $y_P = y_P(t, \vec{s})$. So generally speaking, we have

$$g = g(x_P(t, \vec{s}), y_P(t, \vec{s}), t, \vec{s}) \tag{1.5}$$

In Horn and Schunck's framework [1], $g = g(x_P(t), y_P(t), t)$, as mentioned earlier. This is actually a special case of Eq. (1.5), i.e.,

$$g = g(x_P(t, \vec{s} = \text{const. vector}), y_P(t, \vec{s} = \text{const. vector}), t, \vec{s} = \text{const. vector}),$$

the variation of $\vec{s}$ is restricted to be zero, and the brightness varies temporally only. The UOFF approach is not restricted to any one specific direction; all possible forms assumed by the general brightness function $g(x, y, t, \vec{s})$ construct an imaging space.

We can consider the brightness function as some kind of "density" function. In imaging space, all of the image points corresponding to an arbitrary but fixed point in 3-D world space possess the same brightness, hence the same density. At an arbitrary moment in time, and for different sensor positions (varying Δs), these equal-density image points form an equal-density line. When *time* is varied, this equal-density line changes to another equal-density line. Except if the world point moves out of the scene, these equal-density lines will not disappear nor terminate. The whole imaging space can be viewed consisting of various equal-density lines.

In accordance with the movement in 3-D world space, the distribution of these equal-density lines in imaging space is also varying. In other words, the distribution of these equal-density lines in imaging space reflects the position of objects in 3-D world space, and the *change* of the distribution describes the movement of objects in 3-D world space.

Chapter 2

THEORETICAL DEVELOPMENT

As mentioned earlier, the UOFF approach is a relatively new method that combines both parameters of the brightness function: time, and space. In this chapter, we will go through the development of the general case of the brightness invariance equation, and then solve it for the spatial variation case, in which we are interested.

2.1 Brightness Invariance Equation (BIE)

In the imaging space, all of the image points corresponding to an arbitrary but fixed point P at time t, in 3-D world space possess the same brightness, i.e., P is *isotropical.*

If the optical radiation of a world point P is invariant with respect to a time interval from t_1 to t_2, we then have:

$$g(x_p(t_1, \vec{s}_1), y_p(t_1, \vec{s}_1), t_1, \vec{s}_1) = g(x_p(t_2, \vec{s}_1), y_p(t_2, \vec{s}_1), t_2, \vec{s}_1) \tag{2.1}$$

This is the brightness *time*-invariance equation and is utilized in the determination of optical flow by Horn and Schunck [1]. At a specific moment t_1, if the optical radiation of P is isotropical we then get:

$$g(x_p(t_1,\vec{s}_1),y_p(t_1,\vec{s}_1),t_1,\vec{s}_1) = g(x_p(t_1,\vec{s}_2),y_p(t_1,\vec{s}_2),t_1,\vec{s}_2) \tag{2.2}$$

This is the brightness *space*-invariance equation. If the two variables, time and space, are considered simultaneously, we get the brightness *time-and-space* invariance equation, i.e.,

$$g(x_p(t_1,\vec{s}_1),y_p(t_1,\vec{s}_1),t_1,\vec{s}_1) = g(x_p(t_2,\vec{s}_2),y_p(t_2,\vec{s}_2),t_2,\vec{s}_2) \tag{2.3}$$

Consider two brightness functions $g(x(t,\vec{s}),y(t,\vec{s}),t,\vec{s})$ and $g(x(t+\Delta t,\vec{s}+\Delta\vec{s}),y(t+\Delta t,\vec{s}+\Delta\vec{s}),t+\Delta t,\vec{s}+\Delta\vec{s})$ in which the variation in time, Δt, and the variation in spatial position of sensor, $\Delta\vec{s}$, are very small. Due to the time-and-space-invariance of brightness, we can get:

$$g(x(t,\vec{s}),y(t,\vec{s}),t,\vec{s}) = g(x(t+\Delta t,\vec{s}+\Delta\vec{s}),y(t+\Delta t,\vec{s}+\Delta\vec{s}),t+\Delta t,\vec{s}+\Delta\vec{s}) \tag{2.4}$$

The expansion of the right-hand side of the above equation in the Taylor series leads to

$$\begin{aligned} g(x(t+\Delta t,\vec{s}+\Delta\vec{s}),y(t+\Delta t,\vec{s}+\Delta\vec{s}),t+\Delta t,\vec{s}+\Delta\vec{s}) &= g(x(t,\vec{s}),y(t,\vec{s}),t,\vec{s}) + \\ \frac{\partial g}{\partial x}(\frac{\partial x}{\partial t}\Delta t+\frac{\partial x}{\partial\vec{s}}\Delta\vec{s}) + \frac{\partial g}{\partial y}(\frac{\partial y}{\partial t}\Delta t+\frac{\partial y}{\partial\vec{s}}\Delta\vec{s}) &+ \frac{\partial g}{\partial t}\Delta t+\frac{\partial g}{\partial\vec{s}}\Delta\vec{s}+\epsilon \end{aligned} \tag{2.5}$$

where ϵ contains the second and higher order terms in Δt and/or $\Delta\vec{s}$. The next equation follows then from the use of Eq. (2.3)

$$(\frac{\partial g}{\partial x}u+\frac{\partial g}{\partial y}v+\frac{\partial g}{\partial t})\Delta t + (\frac{\partial g}{\partial x}\vec{u^s}+\frac{\partial g}{\partial y}\vec{v^s}+\frac{\partial g}{\partial\vec{s}})\Delta\vec{s}+\epsilon = 0 \tag{2.6}$$

where $u \triangleq \frac{\partial x}{\partial t}$, $v \triangleq \frac{\partial y}{\partial t}$, $\vec{u^s} \triangleq \frac{\partial x}{\partial\vec{s}}$, $\vec{v^s} \triangleq \frac{\partial y}{\partial\vec{s}}$. Dividing both sides of the above equation by Δt, ignoring the term containing ϵ and examining the limit as $\Delta t \to 0$ yields,

$$\frac{\partial g}{\partial t}+\frac{\partial g}{\partial x}(\frac{\partial x}{\partial t}+\frac{\partial x}{\partial\vec{s}}\frac{\delta\vec{s}}{\delta t})+\frac{\partial g}{\partial y}(\frac{\partial y}{\partial t}+\frac{\partial y}{\partial\vec{s}}\frac{\delta\vec{s}}{\delta t})+\frac{\partial g}{\partial\vec{s}}\frac{\delta\vec{s}}{\delta t}=0 \tag{2.7}$$

where $\frac{\delta\vec{s}}{\delta t} \triangleq \lim_{\Delta t\to 0} \frac{\Delta\vec{s}}{\Delta t}$.

Denote the velocity of a point in the image space by

$$\vec{V} = (\frac{dx}{dt}, \frac{dy}{dt}, \frac{\delta\vec{s}}{\delta t})$$

where $\frac{d}{dt} = \frac{\partial}{\partial t} + \frac{\partial}{\partial\vec{s}}\frac{\delta\vec{s}}{\delta t}$ is a differential operator.

Let $\nabla = (\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial\vec{s}})$ be a vector operator in imaging space. Eq. (2.7) then becomes

$$\frac{\partial g}{\partial t} + \vec{V}\cdot\nabla g = 0 \tag{2.8}$$

Similar to the well-known continuity equation in fluid dynamics [5]:

$$\frac{\partial g}{\partial t} + \nabla\cdot(g\vec{V}) = 0 \tag{2.9}$$

There is a continuity equation in the unified optical flow field which characterizes the brightness invariance [5]. As compared with Eq. (2.9), the left-hand side of Eq. (2.8) lacks a term of $g\nabla\cdot\vec{V}$. It is not difficult to see that the missing term is related to the sum of all the second and higher order terms of Δt and/or $\Delta\vec{s}$, i.e., the ϵ in the right-hand side of Eq. (2.5).

2.2 Unified Optical Flow Field (UOFF)

The brightness time-and-space invariant Eq. (2.6) developed above, is the general case of the UOFF approach, including both the temporal and the spacial variations affecting the brightness function. We will now discuss all the special cases of Eq. (2.6), and then extend our discussion in the direction of interest of this thesis, namely the spatial variation with $\Delta t = 0$.

2.2.1 Special Cases of the BIE

Case 1: If $\Delta\vec{s} = 0$, i.e., the sensor is static in a fixed spatial position, only time varies. Dividing both sides of the equation by Δt and evaluating the limit as $\Delta t \to 0$

degenerate Eq. (2.6) into:

$$\frac{\partial g}{\partial x}u + \frac{\partial g}{\partial y}v + \frac{\partial g}{\partial t} = 0 \tag{2.10}$$

This is the result derived by Horn and Schunck [1].

Case 2: If $\Delta t = 0$, its both sides are divided by $\Delta \vec{s}$ and $\Delta \vec{s} \to 0$ is examined, Eq. (2.6) then reduces to:

$$\frac{\partial g}{\partial x}\vec{u^s} + \frac{\partial g}{\partial y}\vec{v^s} + \frac{\partial g}{\partial \vec{s}} = 0 \tag{2.11}$$

When $\Delta t = 0$, i.e., at a specific time moment, the images generated with sensors at different spatial positions can be viewed as a spatial sequence of images. Eq. (2.11) is then the equation for the spatial sequence of images. It is this equation that we will use for the recovery of depth in the next section, since in our case we have two cameras taking pictures at the same time.

Case 3: If $\frac{\Delta \vec{s}}{\Delta t}$ is a constant, Eq. (2.6) is the equation for a sequence of images taken by a sensor experiencing a uniform motion.

Case 4: If $\frac{\Delta \vec{s}}{\Delta t} = f(\vec{s}, t)$ and the function $f(\vec{s}, t)$ is given, Eq. (2.6) can then be utilized to treat the case when the sensor is experiencing a known movement.

2.2.2 Solving the BIE for the Spatial Case

Reproducing the BIE developed in Case 2 above, we have,

$$\frac{\partial g}{\partial x}\vec{u^s} + \frac{\partial g}{\partial y}\vec{v^s} + \frac{\partial g}{\partial \vec{s}} = 0 \tag{2.12}$$

Let us take a close look at each quantity in the above equation. As mentioned earlier, we are pursuing the *direct* approach for the determination of the depth, and one of the advantages of the direct method is that all of the quantities are readily computable from the image data. The $u^{\vec{s}}$ and $v^{\vec{s}}$ are defined as follows. Let

$$\delta x \triangleq x^R - x^L \qquad\qquad \delta y \triangleq y^R - y^L$$

where (x^R, y^R) and (x^L, y^L) are projections of the same world point on the right and the left image planes, respectively. δx and δy are therefore, respectively, the horizontal and vertical coordinate differences of the image points, corresponding to the same world point in 3-D space, reflected on the right and left image planes. And,

$$u^{\vec{s}} \triangleq \lim_{\delta s \to 0} \frac{\delta x}{\delta s} \tag{2.13}$$

$$v^{\vec{s}} \triangleq \lim_{\delta s \to 0} \frac{\delta y}{\delta s} \tag{2.14}$$

Hence, $u^{\vec{s}}$ and $v^{\vec{s}}$ defined above are the spatial variation rates of δx and δy with respect to $\delta \vec{s}$. These two quantities generated from the *spatial* sequence of images, can be viewed as counterparts of u^L and v^L, (or u^R and v^R), generated from a *temporal* sequence of images.

Following our system setup, the left sensor is located at the origin of our Cartesian coordinate system, and the other sensor is at a known different position. The variation, and hence the variation rate, of the right sensor, from the origin of the world coordinate center, can be decomposed into translational, and rotational components.

We are more interested in the variation rate, rather than the variation itself; this is what is used in the brightness invariance equation. $\vec{T_s}$ is the translational rate, and $\vec{\omega_s}$ the rotational rate. The subscript s indicates the s-domain. We also define (see Fig. 3.6)

$$\vec{T_s} = (U_s, V_s, W_s)^T \tag{2.15}$$

$$\vec{\omega_s} = (A_s, B_s, C_s)^T \tag{2.16}$$

$$\vec{V_s} = \left(\frac{dX}{ds}, \frac{dY}{ds}, \frac{dZ}{ds}\right)^T \tag{2.17}$$

where the superscript T represents the transposition of the concerned vectors. Now, if $\vec{r}$ is defined as the position of an *image* point, and $\vec{r_s}$ is defined as the change in position $\vec{r}$ due to the spatial variation s, then,

$$\vec{V_s} = -\vec{T_s} - \vec{\omega_s} \times \vec{r_s} \tag{2.18}$$

or, in component form:

$$\frac{dX}{ds} = -U_s - B_s Z + C_s Y \tag{2.19}$$

$$\frac{dY}{ds} = -V_s - C_s X + A_s Z \tag{2.20}$$

$$\frac{dZ}{ds} = -W_s - A_s Y + B_s X \tag{2.21}$$

Reproducing the prospective projection formulas we have

$$x = \frac{X}{Z} \qquad\qquad y = \frac{Y}{Z}$$

Combining this with Eq. (2.19), we have:

$$\frac{dX}{ds} = \frac{d}{ds}(xZ) = x\frac{dZ}{ds} + Z\frac{dx}{ds}$$

$$\Rightarrow u^s = \frac{dx}{ds} = \frac{\frac{dX}{ds}}{Z} - \frac{X\frac{dZ}{ds}}{Z^2} \tag{2.22}$$

and similarly with Eq. (2.20):

$$\frac{dY}{ds} = \frac{d}{ds}(yZ) = y\frac{dZ}{ds} + Z\frac{dy}{ds}$$

$$\Rightarrow v^s = \frac{dy}{ds} = \frac{\frac{dY}{ds}}{Z} - \frac{Y\frac{dZ}{ds}}{Z^2} \tag{2.23}$$

From Eqs. (2.22) and (2.19) we get:

$$u_s = (-\frac{U_s}{Z} - B_s + C_s y) - x(-\frac{W_s}{Z} - A_s y + B_s x) \tag{2.24}$$

and from Eqs. (2.23) and (2.20) we get:

$$v_s = (-\frac{V_s}{Z} + A_s - C_s x) - y(-\frac{W_s}{Z} - A_s y + B_s x) \tag{2.25}$$

Substituting Eqs. (2.24) and (2.25) into the brightness invariance equation for the spacial case

$$\frac{\partial g}{\partial x}\vec{u^s} + \frac{\partial g}{\partial y}\vec{v^s} + \frac{\partial g}{\partial \vec{s}} = 0$$

one can obtain

$$\frac{1}{Z} = \frac{(-A_s y + B_s x)(x g_x + y g_y) - g_x(-B_s + C_s y) - g_y(-C_s x + A_s) - g_s}{x W_s g_x + y W_s g_y - U_s g_x - V_s g_y} \tag{2.26}$$

or if we define $Q = 1/Z$,

$$\frac{1}{Z} = Q(x, y, g_x, g_y, g_s, A_s, B_s, C_s, U_s, V_s, W_s) \tag{2.27}$$

Note that $A_s, B_s, C_s, U_s, V_s, W_s$ can be determined once the relative positions of the two sensors in stereo imagery are known. Also, g_x, g_y, g_s can be determined from the given image data by similar algorithms as used in [1]. x, y are the coordinated on the image plane, and thus are known. So now we have a way of recovering the depth, from available image data, the next step would be to use Eq. (2.26) to reconstruct the polynomial describing the surface.

2.3 Surface Structure

The object under study in the 3-D world, is a surface that can be described by a polynomial of degree N, in X, Y, and Z. The general form of the polynomial is

$$\sum_{j=0}^{K-1} \lambda_j X^{\alpha_j} Y^{\beta_j} Z^{\gamma_j} = 0 \tag{2.28}$$

where $0 \leq \alpha_j + \beta_j + \gamma_j \leq N$, and K is the number of coefficients present.

2.3.1 Finding K

For the polynomial representation used in Eq. (2.28), K is the number of coefficients that make up the entire polynomial. It is a function of the degree of the polynomial;

the higher the order, the bigger K becomes. For $N = 1$ or 2, K is easily found by intuition, but as N becomes bigger, it becomes more difficult to determine K manually. For our solution program to be robust, it needs to be able to calculate K readily from the knowledge of N. A procedure to compute K is developed in Appendix A, the final formula is the following

$$K = \sum_{n=0}^{N} \frac{(m+n-1)!}{n!(m-1)!} \tag{2.29}$$

where, N is the order of the polynomial that we want to test for, and m, is the number of variables in the polynomial, in our case, 3: $(X, Y,$ and $Z)$.
This formula is used at the beginning of the program for the computation of K knowing $m = 3$, and N.

2.3.2 Polynomial Normalization

As seen earlier, the surface under consideration is described by the polynomial

$$\sum_{j=0}^{K-1} \lambda_j X^{\alpha_j} Y^{\beta_j} Z^{\gamma_j} = 0$$

Out of the K different coefficients of the polynomial, there are $K-1$ independent terms, the polynomial can therefore be normalized with respect to one arbitrary coefficient $\lambda(r)$.
By rewriting the polynomial as

$$\sum_{j=0 j\neq r}^{K-1} \left(\lambda_j X^{\alpha_j} Y^{\beta_j} Z^{\gamma_j}\right) + \lambda_r X^{\alpha_r} Y^{\beta_r} Z^{\gamma_r} = 0,$$

we divide through by term λ_r,

$$\sum_{j=0 j\neq r}^{K-1} \left[\lambda_{nj} X^{\alpha_j} Y^{\beta_j} Z^{\gamma_j}\right] + X^{\alpha_r} Y^{\beta_r} Z^{\gamma_r} = 0 \tag{2.30}$$

where $\lambda_{nj} = \frac{\lambda_j}{\lambda_r}$, the normalized λ_j. This equation will be used in reconstructing the polynomial since now we have $(K-1)$ independent coefficients.

2.4 Reconstructing the Polynomial

Substituting Q^{-1} for Z n Eq. (2.30), we get:

$$\sum_{j=0j\neq r}^{K-1}\left[\lambda_{nj}X^{\alpha_j}Y^{\beta_j}Q^{-(\gamma_j)}\right]+X^{\alpha_r}Y^{\beta_r}Q^{-(\gamma_r)}=0 \tag{2.31}$$

2.4.1 A Least Squares Formation

Now, to reconstruct the original polynomial we have to use the recovered depth as in Eq. (2.31) so we define a performance function as

$$\wp=\int\int_{\Re}\{\sum_{j=0j\neq r}^{K-1}\left[\lambda_{nj}X^{\alpha_j}Y^{\beta_j}Q^{-(\gamma_j)}\right]+X^{\alpha_r}Y^{\beta_r}Q^{-(\gamma_r)}\}^2dxdy \tag{2.32}$$

where $\Re$ is the region on the image plane associated with the concerned surface in 3-D space. The task here is to find a set of coefficients λ_{nj} so that the performance function $\wp$ is minimized (brought as close to zero as possible). It is well known that the following linear equations are necessary conditions for minimization of the $\wp$ function:

$$\frac{\partial\wp}{\partial\lambda_i}=0 \tag{2.33}$$

where $i=0,1,2,\ldots,(K-1)$, and $i\neq r$. Differentiating with respect to λ_i yields

$$\int\int_{\Re}2\{\sum_{j=0j\neq r}^{K-1}\left[\lambda_{nj}X^{\alpha_j}Y^{\beta_j}Q^{-(\gamma_j)}\right]+X^{\alpha_r}Y^{\beta_r}Q^{-(\gamma_r)}\}\{X^{\alpha_i}Y^{\beta_i}Q^{-\lambda_i}\}dxdy=0 \tag{2.34}$$

or,

$$\sum_{j=0j\neq r}^{K-1}\left[\int\int_{\Re}\left(X^{\alpha_j+\alpha_i}Y^{\beta_j+\beta_i}Q^{-(\gamma_j+\gamma_i)}\right)dxdy\right]\lambda_j= -\int\int_{\Re}X^{\alpha_r+\alpha_i}Y^{\beta_r+\beta_i}Q^{-(\gamma_r+\gamma_i)} \tag{2.35}$$

with $i=0,1,2,\ldots,(K-1)$, and $i\neq r$.

2.4.2 A Set of Linear Equations

The above equations can be put in matrix form as follows:

$$M_{i,j} = \left[\int\int_{\Re} \left(X^{\alpha_j+\alpha_i} Y^{\beta_j+\beta_i} Q^{-(\gamma_j+\gamma_i)}\right) dx dy\right] \tag{2.36}$$

$$D_i = -\int\int_{\Re} X^{\alpha_r+\alpha_i} Y^{\beta_r+\beta_i} Q^{-(\gamma_r \gamma_i)} \tag{2.37}$$

In this set of linear equations, all of the coefficients of the Nth degree polynomial, i.e., λ_i, where $i = 0, 1, 2, \ldots, (K-1)$, and $i \neq r$ are unknown. All of the entries in the matrix $M_{i,j}$ and in the vector D_i can be computed from the given image data.

The structure of the surface can therefore be recovered, because the polynomial equation describing the surface has been fully determined.

Chapter 3

SIMULATION AND RESULTS

To test the previous algorithm for recovering surface structure, a computer program was written in C. The program can be divided into three major parts:

1. The first part is to build a pair of stereo images of a surface (Sphere, or a 3-rd order Alpha shape, choice is user's), and save them in image files of variable sizes (typically 128 by 128).

2. The second part, which knows nothing about the first one, deals only with the image data created earlier, and attempts to recover the depth field at each image pixel, by applying Eq. (2.26).

3. The final part uses the recovered depth, and sets up the M, and D matrices, and then solves to recover the $(K-1)$ independent λ's.

3.1 The Simulation Images

We start with the first part, which assumes a certain polynomial structure and performs a prospective projection from the surface onto the two image planes, the left and the right.

3.1.1 The Polynomials

Two different surfaces have been considered in our study: a Sphere surface, and an Alpha shape surface.

The Sphere

The Sphere chosen is described by the following polynomial:

$$X^2 + Y^2 + (Z - D)^2 - 16 = 0 \tag{3.1}$$

or in the shifted coordinates (see Fig. 3.5):

$$X'^2 + Y'^2 + Z'^2 - 16 = 0 \tag{3.2}$$

The Alpha Shape

The Alpha shape surface used is described by

$$Y'^3 - 5Y'^2 + X'^2 + Z'^2 = 0 \tag{3.3}$$

Both of these two surfaces clearly fit into the general definition of the polynomial given by

$$\sum_{j=0}^{K-1} \lambda(j) X^{\alpha_j} Y^{\beta_j} Z^{\gamma_j} = 0.$$

It is easier to work with the shifted set of world coordinates, as will be explained in the next chapter.

One thing remains to be set however; to associate combinations of $\alpha_j, \beta_j, \gamma_j$, with combinations of X, Y, Z. The standard shown in Table (3.1) is the one assumed throughout our work. Any arbitrary combination will do, as long as it is not changed in the middle of the procedure, however, this one was chosen because it is structured in nature, and because the higher degree terms can e added to later without changing the lower ones.

λ	α	β	γ
λ_0	0	0	0
λ_1	1	0	0
λ_2	0	1	0
λ_3	0	0	1
λ_4	2	0	0
λ_5	1	1	0
λ_6	1	0	1
λ_7	0	2	0
λ_8	0	1	1
λ_9	0	0	2
λ_{10}	3	0	0
λ_{11}	2	1	0
λ_{12}	2	0	1
λ_{13}	1	2	0
λ_{14}	1	1	1
λ_{15}	1	0	2
λ_{16}	0	3	0
λ_{17}	0	2	1
λ_{18}	0	1	2
λ_{19}	0	0	3

Table 3.1: The Standard α, β, γ Used

From the table, we then know that for the Sphere: $\lambda_0 = -16.0$, $\lambda_4 = 1.0$, $\lambda_7 = 1.0$, $\lambda_9 = 1.0$.

And for the Alpha shape surface:

$\lambda_4 = 1.00$, $\lambda_7 = -5.00$, $\lambda_9 = 1.00$, $\lambda_{16} = 1.00$.

The rest are zeros. If we choose to normalize about $\lambda(4)$, in both cases, then the solution that we should expect from the program would be exactly the same as above.

3.1.2 Building the Images

Both surfaces are built in a similar manner. We start from the screen, project a beam of light toward the direction of the surface, and try to solve for the *closest* point in space in which the beam actually hits the surface. If no solution exists at all, then that particular pixel does not "see" the surface, and is assigned a background gray level value. If solutions do exist, we choose the closest one to the image plane, and then from the knowledge of the coordinates in space of that point, we assign a gray level value according to a generating function. The difference for the two shapes lies in solving for the beam with the structure in space.

The polynomials describing the surfaces are actually in X', Y', Z', the shifted world coordinate system. It is easier to deal with the shifted system as far as describing the surface is concerned.

To associate a gray level with the surface, it is much more practical to convert the Cartesian coordinates into spherical coordinates as follows (see Fig. 3.6):

$$\theta = \tan^{-1}\left(\frac{\sqrt{X'^2 + Z'^2}}{Y'}\right)$$

$$\phi = \tan^{-1}\left(\frac{Z'}{X'}\right)$$

and based on θ and ϕ, the gray level function is assigned.
After a considerable time of experimentation, the following gray function was chosen:

$$g = K_1 \cos(K_2\theta)\sin(K_3\phi) + K_1 + \Delta \tag{3.4}$$

where $K_1 = 2048.0$, and $\Delta = 25.0$. $K2$ and $K3$ are chosen later on a trial-and-error basis, to see which combination gives the best results.
This gray function is a continuous one, however the one saved in the image files is a quantized version of this one, which will produce some quantization error to be

discussed later.

For both surfaces, there is a difference in the way the left sensor image and the right sensor image, are created. These differences are discussed here.

The Left Image

The left sensor is aligned with the world coordinate system, and thus it is relatively a simple matter to project world points onto the screen. The distance D between the origin of the world coordinate system and the origin of the *shifted* coordinate system is chosen to be 100.0 units(meters or feet, etc.) The same length units are assumed for the surface, so we have a relative idea about the size of the surfaces being considered.

To maximize efficiency, the light from the object should occupy about 75% of the screen; so the screen size is taken to be 0.11×0.11 length units. The screen is made of $I_n \times J_n$ pixels. i and j are used as row, and, column indices of the screen. x and y are related linearly to i and j as follows:

$$x = \frac{0.11j}{J_n} + x_0 \qquad y = \frac{-0.11i}{I_n} + y_0 \tag{3.5}$$

where for the sphere, $x_0 = -0.055$ is the offset in the x direction, and, $y_0 = 0.055$ is the offset in the y direction, and for the Alpha shape, $y_0 = 0.07$. Note that x moves in the same direction of j, while y moves in the opposite direction of i. Note also that for the Alpha shape image, the $y = 0$ line is not evenly placed in the screen, this is because the surface considered expands very rapidly for negative values of y. If we include a larger portion of the negative y axis in the picture, either the picture will go out of the screen boundaries distorting the image information, or we will be forced to distance the camera more from the object, thus reducing considerably the size of the portion of the image corresponding to the positive y axis.

In building the left image, we have to start from the *screen,* not from the world point, and our Algorithm goes as follows: For a point $p(i,j)$ on the screen, x, and y are calculated as in Eq. (3.5). The surface equation for the Alpha shape surface in the *world* coordinates is

$$Y^3 - 5Y^2 + X^2 + Z^2 - 2DZ + D^2 = 0 \tag{3.6}$$

remember, D here is the distance between the origin of the world coordinate system and the origin og the shifted coordinate system. Using prospective projection, we replace X and Y, with xZ and yZ, respectively, and substitute them into Eq. (3.6) above to obtain a 3rd order equation with one unknown:

$$(y^3)Z^3 + (x^2 - 5y^2 + 1)Z^2 - (2D)Z + (D^2) = 0 \tag{3.7}$$

which can be very easily solved numerically to get Z.
When Z is obtained, we get $Z' = (Z - D), Y' = yZ$, and $X' = xZ$. From X', Y', Z', we get θ and ϕ as shown above, and we obtain the gray level value from the function. The quantized value of g is then assigned to the pixel at (i,j) which we started with.

This is repeated for all the screen pixels, and if at some pixel no reasonable solution is obtained for Z, the program then knows that this pixel does not "see" the image, and assigns a background gray value to it.

The Right Image

The right sensor's position in 3-D space differs from that of the left sensor by a mere rotation of angle β about the $O'Y'$ axis. The angle is positive when viewed from the positive Y' axis down onto the origin and moves clockwise. The arm of rotation is D, and thus the distance between the camera and the world origin is kept the same. Furthermore, the relation between i and j, and x^R and y^R remains the same, namely,

$$x^R = \frac{0.11j}{J_n} + x_0 \qquad\qquad y^R = \frac{-0.11i}{I_n} + y_0$$

Note that the surface is symmetric about the $O'Y'$ axis, the axis of rotation. Therefore, since both sensors stay coplanar, no change in the shape will take place in the right camera.

The difference will come from the gray function values. The gray function will have to be rotated in the opposite direction for our simulation to be correct. So the same procedure is followed here to get X', Y', Z', but in the gray function ϕ is replaced with $(\phi + \beta)$. Following exactly the same procedure from here on, we build the right sensor image and save it in an image file.

3.2 Recovering the Depth

We now have the two stereo images needed for the analysis, the next step is to use the image data to recover the depth field. That is done by using Eq. (2.26):

$$\frac{1}{Z} = \frac{(-A_s y + B_s x)(x g_x + y g_y) - g_x(-B_s + C_s y) - g_y(-C_s x + A_s) - g_s}{x W_s g_x + y W_s g_y - U_s g_x - V_s g_y}$$

The direct method, as mentioned earlier, can recover directly the depth from the image data. We are going to examine the terms that make up Eq. (2.26), and see how we can obtain all of them.

3.2.1 x and y

Q, is an array of the same size of the image screen, i.e. I_n by J_n. The depth Z, in other words, is computed for all the pixels of the screen. The solution program deals with image pixels and not with continuous values of x and y. If we denote i

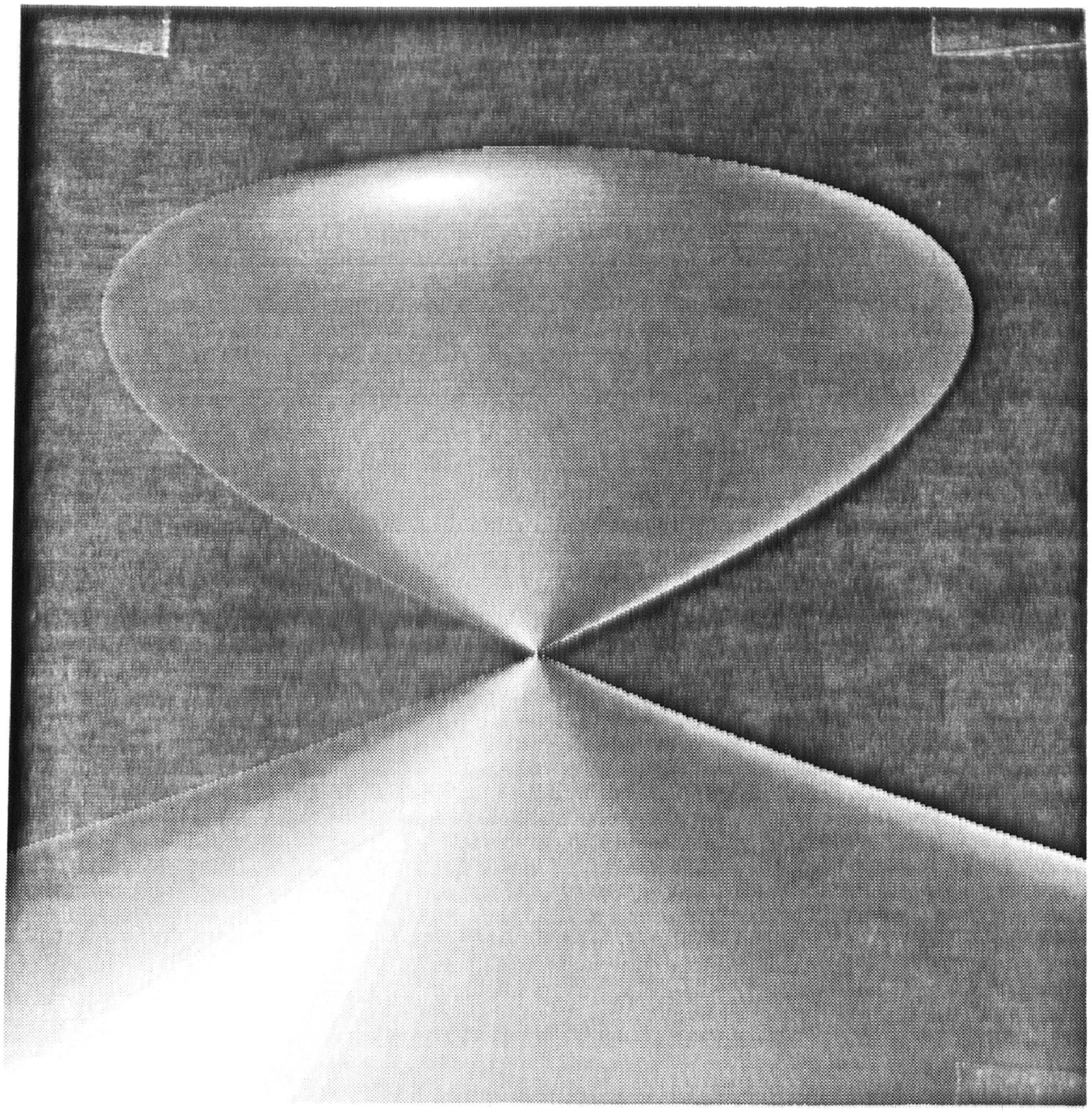

Figure 3.1: Alpha Shape Surface Seen from the Left Camera

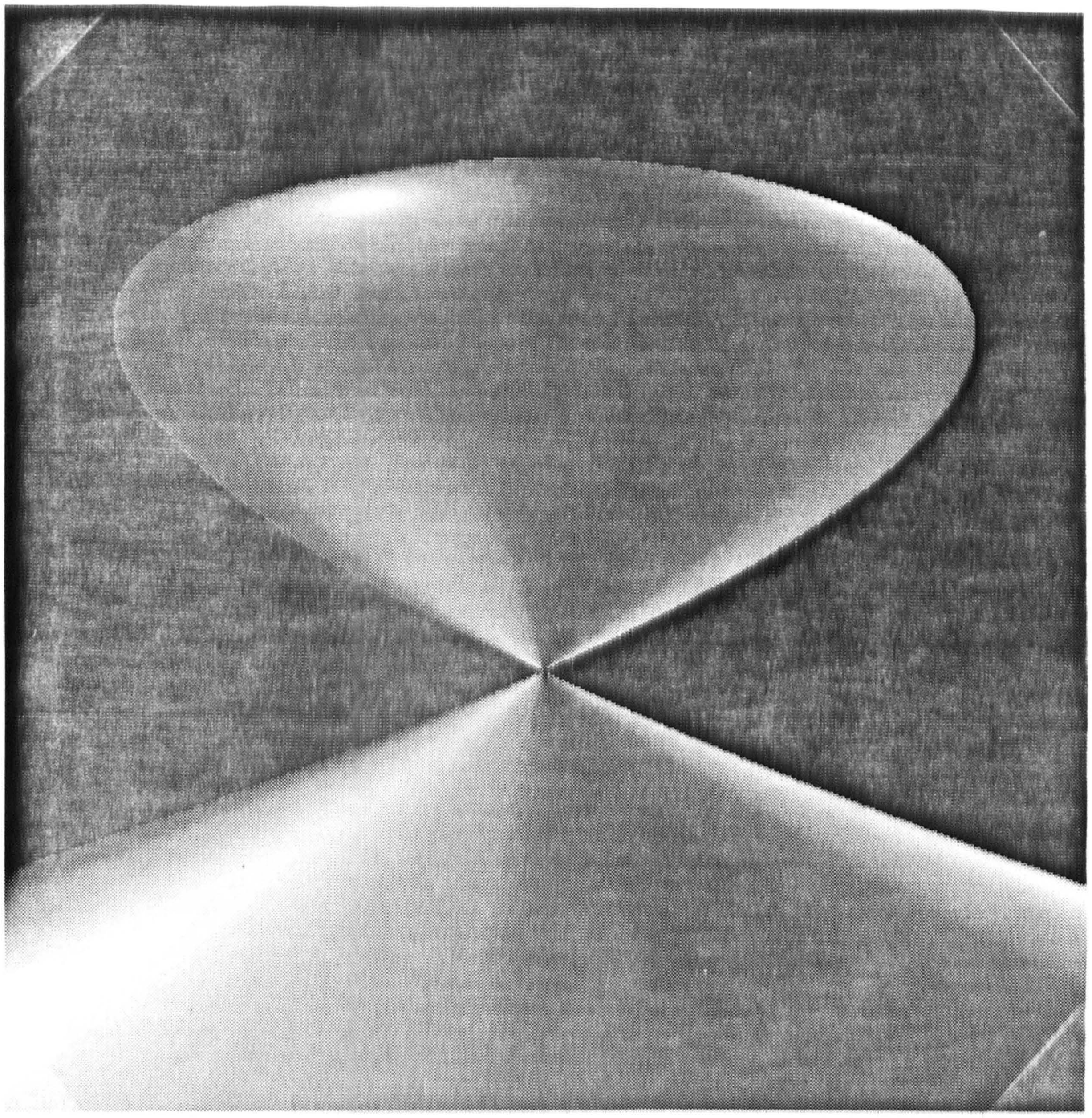

Figure 3.2: Alpha Shape Surface Seen from the Right Camera ($\beta = 15°$)

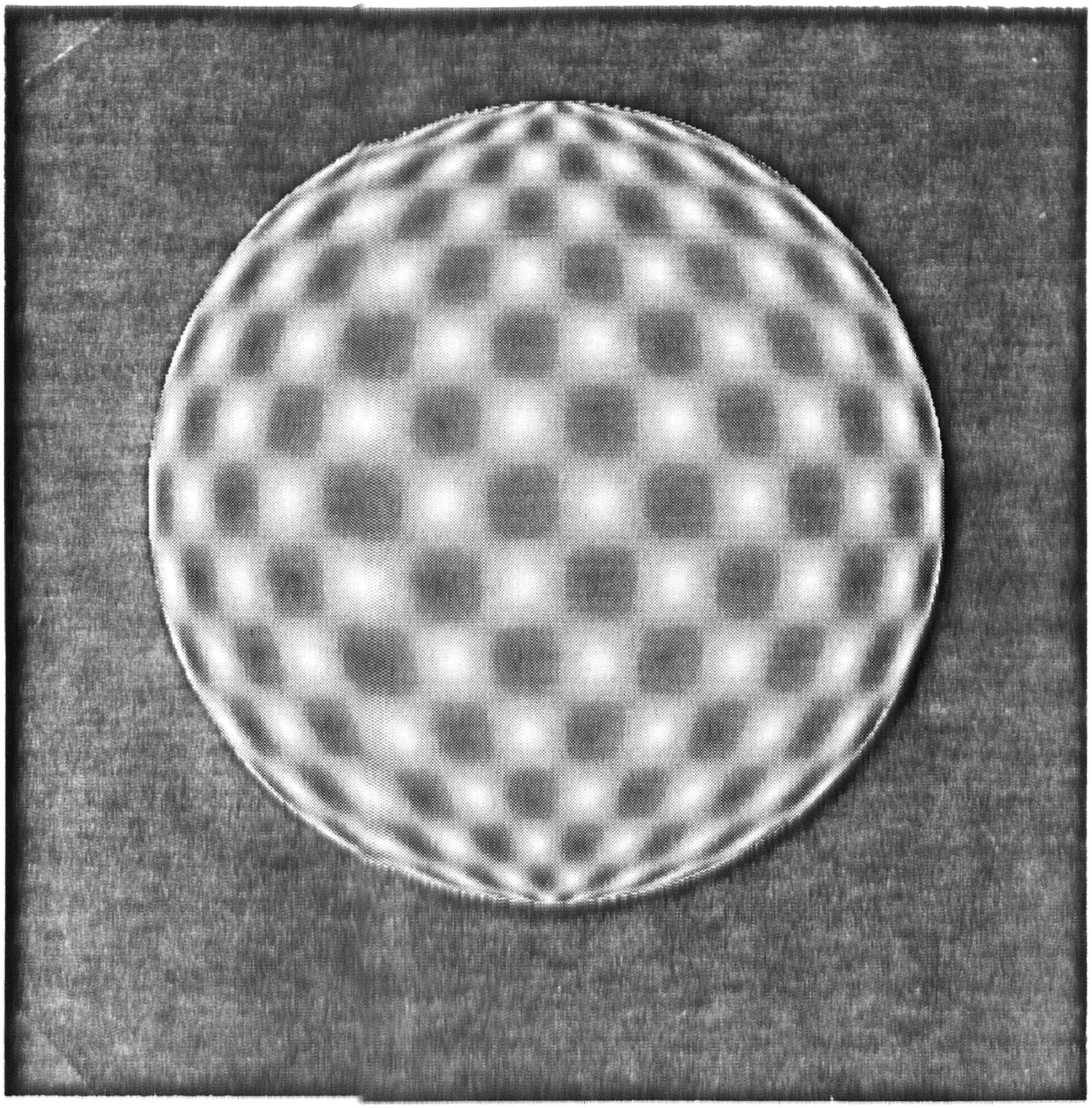

Figure 3.3: Sphere Surface Seen from the Left Camera

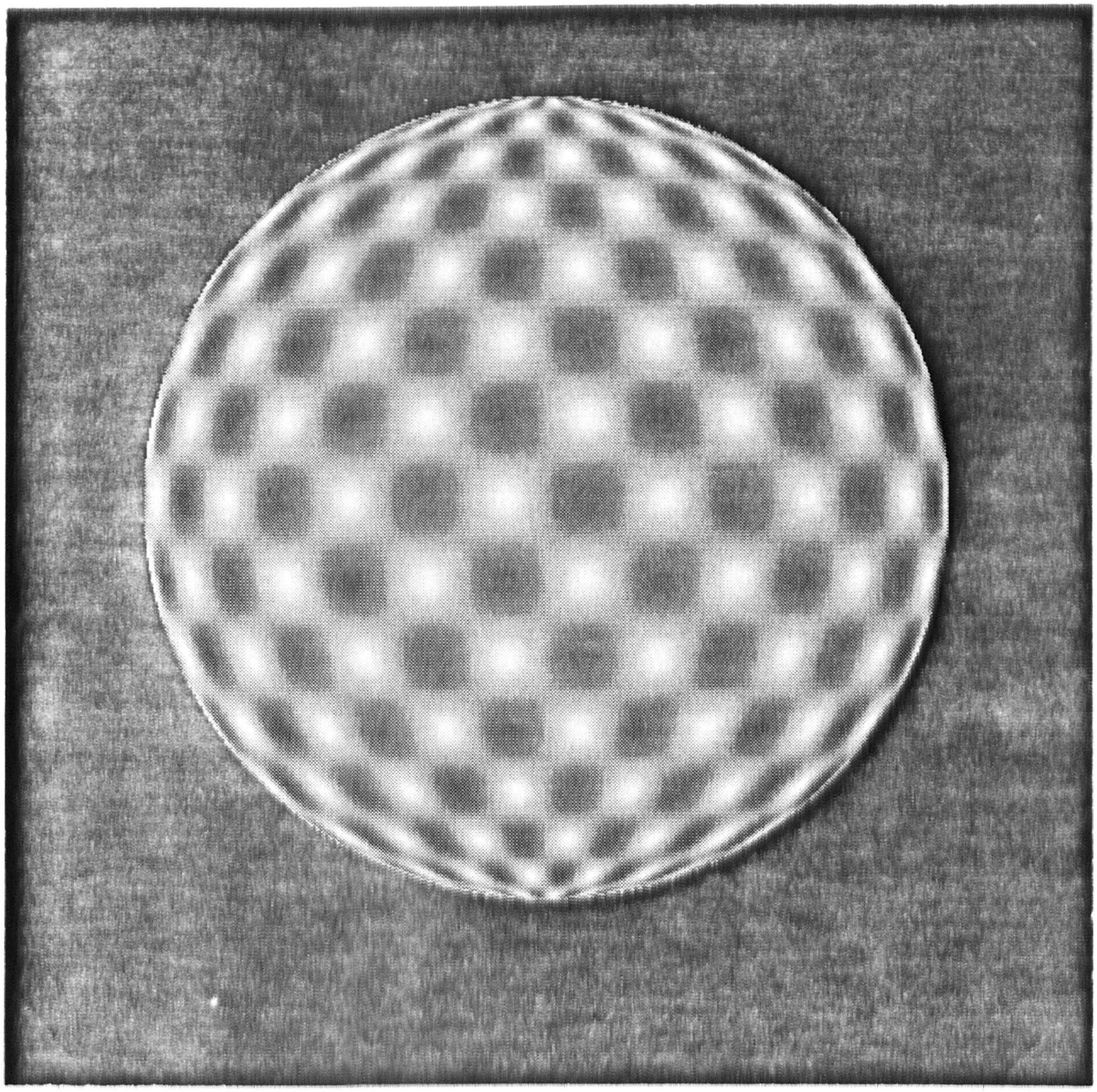

Figure 3.4: Sphere Surface Seen from the Right Camera ($\beta = 10°$)

to represent the row index, and j to represent the column index, then we have, as seen earlier, a linear relation between the indices and the real values. x and y in Eq. (2.26), are calculated as shown in Eq. (3.5), or,

$$x = \frac{0.11j}{J_n} + x_0 \qquad\qquad y = \frac{-0.11i}{I_n} + y_0$$

3.2.2 Gray Level Gradients

Reproducing the brightness invariance equation (2.12) we have,

$$\frac{\partial g^L}{\partial x}\vec{u^s} + \frac{\partial g^L}{\partial y}\vec{v^s} + \frac{\partial g^L}{\partial \vec{s}} = 0$$

For short, we have referred to $\frac{\partial g^L}{\partial x}$, $\frac{\partial g^L}{\partial y}$, and $\frac{\partial g^L}{\partial \vec{s}}$ as g_x, g_y, and, g_s, respectively.

Obtaining g_x and g_y

Horn and Scunck in [1] describe a simple approximation of g_x and g_y. We are interested in computing the x and y gradients for a spatial sequence of images, not a temporal one. The approximations from [1] are therefore modified slightly:

$$\Delta g_j = \frac{1}{4}\{L(i,j+1) - L(i,j) + L(i+1,j+1) - L(i+1,j)$$

$$+ R(i,j+1) - R(i,j) + R(i+1,j+1) - R(i+1,j)\} \qquad (3.8)$$

$$\Delta g_i = \frac{1}{4}\{L(i+1,j) - L(i,j) + L(i+1,j+1) - L(i,j+1)$$

$$+ R(i+1,j) - R(i,j) + R(i+1,j+1) - R(i,j+1)\} \qquad (3.9)$$

where $L(i,j)$ and $R(i,j)$, are the gray values of the pixels at row i and column j, of the left, and right images, respectively. Considering the change in the values of x, and y, from pixel to pixel,

$$\Delta x_j = \frac{0.11}{I_n} \qquad \Delta y_i = \frac{-0.11}{J_n} \tag{3.10}$$

where I_n and J_n are the number of pixels in the row and the column, respectively. This way, we can approximate $\frac{\partial g^L}{\partial x}, \frac{\partial g^L}{\partial y}$ by:

$$\frac{\partial g}{\partial x} \approx \frac{\partial g}{\partial j}\frac{\partial j}{\partial x} \approx \frac{\Delta g_j}{\Delta x_j} \tag{3.11}$$

$$\frac{\partial g}{\partial y} \approx \frac{\partial g}{\partial i}\frac{\partial i}{\partial y} \approx \frac{\Delta g_i}{\Delta y_i} \tag{3.12}$$

Obtaining g_s

The same reference as above [3], provides an approximation for g_s, which we use here:

$$\Delta g_s = R(i,j) - L(i,j) + R(i+1,j) - L(i+1,j)$$

$$+ R(i,j+1) - L(i,j+1) + R(i+1,j+1) - L(i+1,j+1) \tag{3.13}$$

Now, g_s is a *rate* of change of spatial gradient, so it has to be divided by an argument that contains a measure of the transition between the left and right cameras; the spatial transition.

$$\delta\vec{s} \triangleq \sqrt{\tilde{x}^2 + \tilde{z}^2 + D^2\beta^2}$$

where $\tilde{x}$ and $\tilde{z}$, represent the displacement of the *right* optical center from that of the left optical center. $D\beta$ is the length of the arc made by the rotation of the camera. In short, $\delta\vec{s}$ is a measure of the *movement* from the first camera to the second, the "Spatial" movement. g_s is therefore approximated as:

$$g_s \approx \frac{\Delta g_s}{\delta\vec{s}} \tag{3.14}$$

3.2.3 Translation and Rotation Rates

In Eq. (2.26), A_s, B_s, C_s, are the components of the rotation rate vector of the right camera in the X, Y, and, Z directions, respectively (see Fig. 3.5). U_s, V_s, and, W_s, are the components of the translation rate vector of the right camera in the X, Y, and, Z directions, respectively. For a setup like the one shown in Fig. 3.5, the values of these components are as follows:

$$\begin{array}{rcl} A_s & = & 0.0 \\ B_s & = & \frac{-\beta}{\delta\vec{s}} \\ C_s & = & 0.0 \\ U_s & = & \frac{D\sin\beta}{\delta\vec{s}} \\ V_s & = & 0.0 \\ W_s & = & \frac{D(1-\cos\beta)}{\delta\vec{s}} \end{array} \tag{3.15}$$

It is seen that all the terms in Eq. (2.26) can be computed from available image data, and hence, the depth can be recovered by this method, as a first step to reconstruct the surface polynomial. Note here, that $\delta\vec{s}$ can be factored out from both the numerator, and the denominator of Eq. (2.26), since all of the added terms contain it. Factoring $\delta\vec{s}$ out makes the calculations faster, especially that these calculations will have to be performed a huge number of times. What this factoring means is a normalization in the direction of $\vec{s}$ in the spatial case, or t in the temporal case.

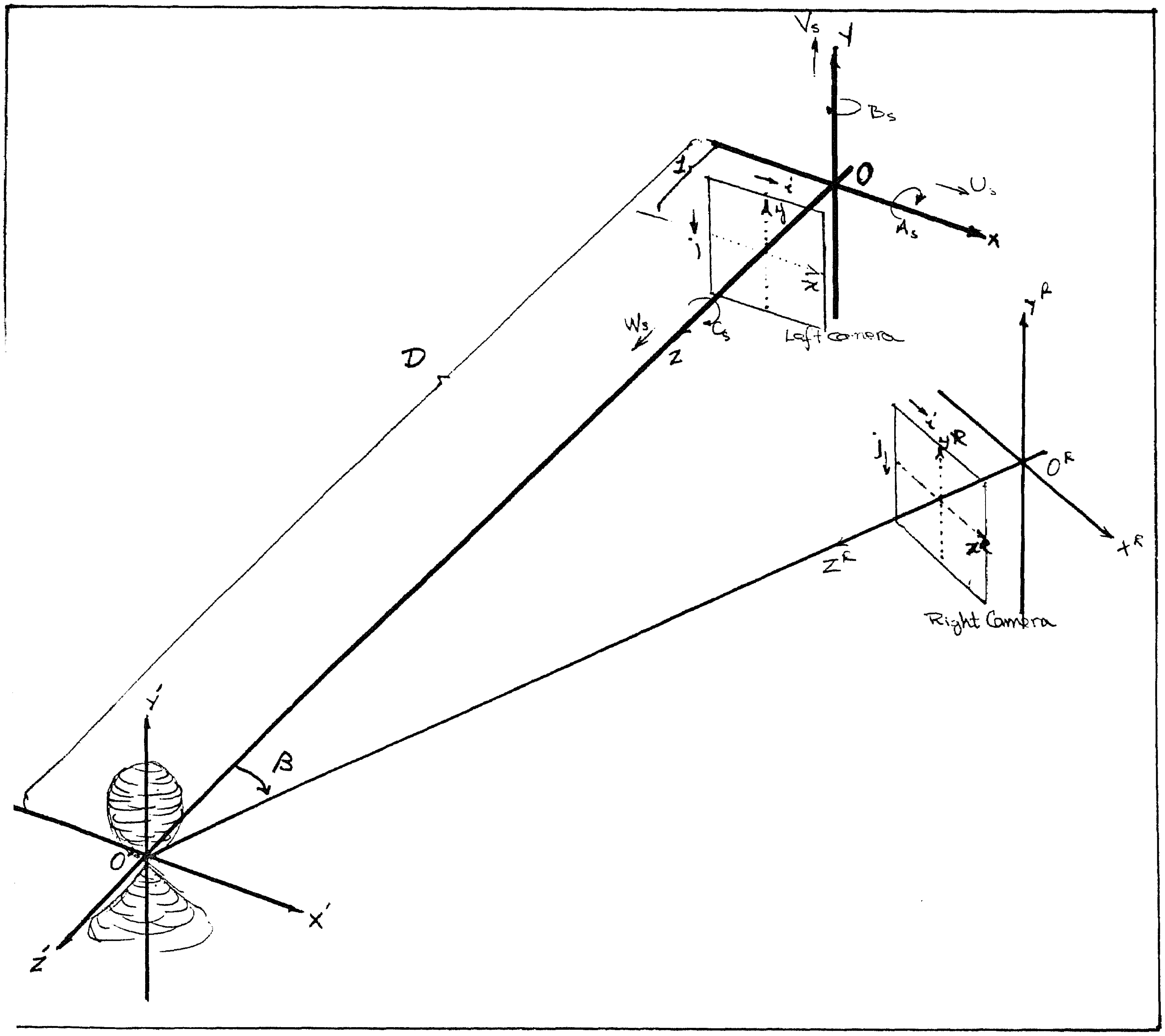

Figure 3.5: System Setup

3.3 Evaluating the Set of Linear Equations

Now, that we've recovered the depth, the next step is to optimize our performance function and estimate the polynomial coefficients.

3.3.1 Numerical Considerations

For a sum of reasons, discussed in the next chapter, solving for $M_{i,j}$ and D_i as defined in Eq. (2.36) and Eq. (2.37), did not give good results. Briefly, the value of Q^{-1} is very large compared with the values of X and Y, so the significance of the values of X and Y was being lost. It was needed to rewrite the performance function in a way that would eventually prevent, or reduce, multiplications of numbers that are too small, or too large, compared to one another. Restarting from Eq. (2.30), and describing the surface in the X', Y', Z' system, we have

$$\sum_{j=0 j\neq r}^{K-1} \left[\lambda_n(j) X'^{\alpha_j} Y'^{\beta_j} Z'^{\gamma_j}\right] + X'^{\alpha_r} Y'^{\beta_r} Z'^{\gamma_r} = 0 \tag{3.16}$$

Going back to the X, Y, Z system, and replacing $(X', Y', Z')^T$ by $(X, Y, (Z-D))^T$, and then putting that into the above equation and substituting Q^{-1} for Z, we get:

$$\sum_{j=0 j\neq r}^{K-1} \left[\lambda_{nj} X^{\alpha_j} Y^{\beta_j} (Q^{-1} - D)^{\gamma_j}\right] + X^{\alpha_r} Y^{\beta_r} (Q^{-1} - D)^{\gamma_r} = 0. \tag{3.17}$$

where $X = xZ$, and $Y = yZ$.

Our performance function is then defined as:

$$\wp = \int\int_{\Re} \{ \sum_{j=0 j\neq r}^{K-1} \left[\lambda_{nj} X^{\alpha_j} Y^{\beta_j} (Q^{-1} - D)^{\gamma_j}\right] + X^{\alpha_r} Y^{\beta_r} (Q^{-1} - D)^{\gamma_r} \}^2 dx dy \tag{3.18}$$

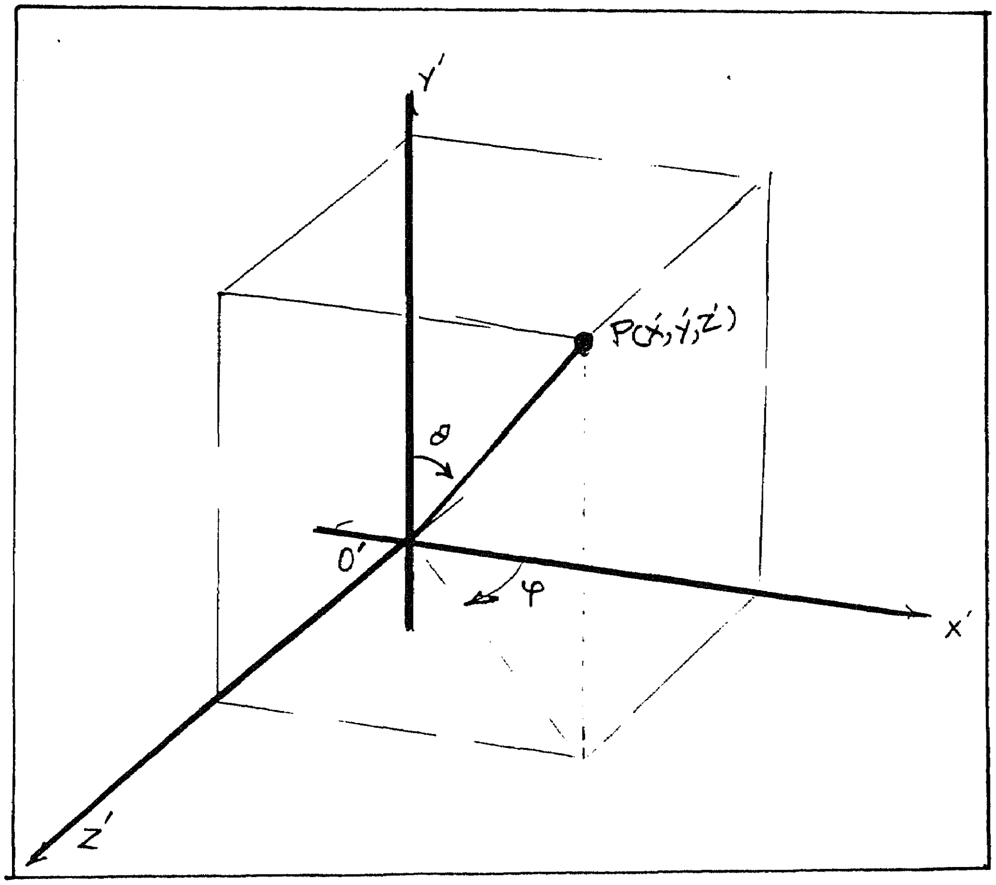

Figure 3.6: The Surface in X', Y', Z' Coordinates

So, our M matrix becomes:

$$M_{i,j} = \int\int_{\Re} \left[X^{(\alpha_j+\alpha_i)} Y^{(\beta_j+\beta_i)} (Q^{-1} - D)^{(\gamma_j+\gamma_i)}\right] \tag{3.19}$$

and, the D vector:

$$D_i = -\int\int_{\Re} \left[X^{(\alpha_r+\alpha_i)} Y^{(\beta_r+\beta_i)} (Q^{-1} - D)^{(\gamma_r+\gamma_i)}\right] \tag{3.20}$$

By putting the equations in this form, the numbers being multiplied are compatible, and will eliminate the type of numerical error that would have been present otherwise.

3.3.2 Computer Algorithm

For the computer, the above integrations in the computation of M and D, are performed as summations. So the computer calculates M and D as follows

$$M_{i,j} = \sum_{i=0}^{(I_n-1)} \sum_{j=0}^{(J_n-1)} \left[X^{(\alpha_j+\alpha_i)} Y^{(\beta_j+\beta_i)} (Q^{-1} - D)^{(\gamma_j+\gamma_i)}\right] \tag{3.21}$$

$$D_i = -\sum_{i=0}^{(I_n-1)} \sum_{j=0}^{(J_n-1)} \left[X^{(\alpha_r+\alpha_i)} Y^{(\beta_r+\beta_i)} (Q^{-1} - D)^{(\gamma_r+\gamma_i)}\right] \tag{3.22}$$

We have seen earlier how all of the terms in the equations above can be obtained.

3.4 Results

Many parameters affect the accuracy of the reconstruction, as will be discussed shortly. The following is a presentation of the best results obtained, for both surfaces, and their associated parameters.

3.4.1 The Alpha Shape Surface

The following data was obtained for $k1 = 4096.0$, $k2 = 0.0$, $k3 = 3.6$.
The size of the image was 64×64.
The rotation angle of the right camera $\beta = 2.0$ degrees.
Normalization is done with respect to λ_4.

n	Term	λ_n
λ_0	1	-0.012383080956
λ_1	X	0.09040302193
λ_2	Y	0.09594640335
λ_3	Z	0.11946278846
λ_4	X^2	**1.00000000000**
λ_5	XY	0.04575281194
λ_6	XZ	-0.04139833537
λ_7	Y^2	**-4.63054524910**
λ_8	YZ	-0.07763573426
λ_9	Z^2	**1.046629711797**
λ_{10}	X^3	-0.00236808628
λ_{11}	X^2Y	-0.02518825026
λ_{12}	X^2Z	0.04122164258
λ_{13}	XY^2	0.00619991298
λ_{14}	XYZ	0.01708973018
λ_{15}	XZ^2	-0.00686164899
λ_{16}	Y^3	**0.89073972725**
λ_{17}	Y^2Z	-0.05525919378
λ_{18}	YZ^2	-0.09711923003
λ_{19}	Z^3	0.04964413011

Table 3.2: Results for the Alpha Shape Reconstruction

The numbers in boldface type, are the ones we are looking for: $\lambda_4, \lambda_7, \lambda_9$, and λ_{16}. We can see that the reconstruction error is bounded to within12% on the maximum. The other λ's, should all be zero. We see some of them not very close to zero such as λ_2, λ_3, and λ_8. Compared with the *smallest* non-zero coefficients, which equal 1.00, the errors in the *zero-coefficients* are bounded to about 15%.

The sources of these errors will be discussed in the next chapter.

3.4.2 The Sphere Surface

Since the sphere is a *second* order surface, it was much more robust to the various errors present. The M matrix here is 9×9, instead of 19×19 for the Alpha shape. This, not only made it almost 500% faster to test than the Alpha shape, for an equal size image file, but also made it easier to detect where the errors are originating from. The discussion of the sources of error is a part of the next chapter.

The following data was obtained for $k1 = 4096.0$, $k2 = 0.0$, $k3 = 0.7500$.
The size of the image was 64×64.
The rotation angle of the right camera $\beta = 1.475$ degrees.

Normalization is done about λ_4.

n	λ_n
λ_0	**-15.9990001671**
λ_1	0.0913221380
λ_2	-0.0406793652
λ_3	-0.0491134118
λ_4	**1.0000000000**
λ_5	0.0042998058
λ_6	**0.9975087321**
λ_8	0.0222900959
λ_9	**1.0138346650**

Table 3.3: Results for the Sphere Shape Reconstruction

Again, the numbers that are in boldface in Table (3.3) are the non-zero coefficients. It is seen here that the maximum error is bounded to **1.00** % !

3.4.3 Multiple-Run Results

The following are samples of multiple-run applications of the program, to demonstrate the sensitivity to the various parameters.

λ **versus** k_3

The following data was obtained for $k_1 = 4096.0$, $k_2 = 0.0$
The size of the image was 64×64.
The rotation angle of the right camera $\beta = 1.400$ degrees.
Normalization is done about λ_4.

k_3	λ_0	λ_4	λ_7	λ_9
0.6500	-15.773061	1.000000	0.990632	1.033322
0.6600	-15.796425	1.000000	0.991945	1.031614
0.6700	-15.803293	1.000000	0.992367	1.030295
0.6800	-15.825661	1.000000	0.993266	1.027575
0.6900	-15.826480	1.000000	0.994136	1.031076
0.7000	-15.854684	1.000000	0.994858	1.025469
0.7100	-15.864268	1.000000	0.995469	1.023938
0.7200	-15.883771	1.000000	0.995847	1.021146
0.7300	-15.885682	1.000000	0.996210	1.022712
0.7400	-15.909165	1.000000	0.996516	1.019486
0.7500	-15.925507	1.000000	0.996118	1.017167
0.7000	-15.947952	1.000000	0.997057	1.014855
0.7700	-15.928014	1.000000	0.997377	1.020163
0.7000	-15.912178	1.000000	0.997469	1.019900
0.7900	-18.757997	1.000000	0.947441	0.329484
0.8000	-19.337301	1.000000	0.931492	0.179540

Table 3.4: Effect of k_3 on the Sphere Shape Solutions

Note that in Table (3.4), for some values of $k_3, (k_3 = 0.8)$, the results are highly erroneous. For most of the remaining runs versus k_3, the errors are bounded to $\pm 5\%$.

λ versus β

The following data was obtained for $k_1 = 4096.0$, $k_2 = 0.0, k_3 = 0.7500$
The size of the image was 64×64.
Normalization is done about λ_4.

β	λ_0	λ_4	λ_7	λ_9
0.1000	-17.005822	1.000000	0.789060	-0.19465
0.3500	-16.442779	1.000000	0.960108	0.758533
0.4750	-16.280832	1.000000	0.986284	0.865729
0.6000	-16.128859	1.000000	0.987973	0.934429
0.7250	-16.223021	1.000000	0.988703	0.911685
0.8500	-16.052388	1.000000	0.992791	0.974399
0.9750	-16.062600	1.000000	0.996652	0.976688
1.1000	-16.018273	1.000000	0.999690	0.997282
1.2250	-16.053506	1.000000	0.996623	0.984401
1.3500	-15.981759	1.000000	0.996400	1.005005
1.4750	-15.999000	1.000000	0.996888	0.997988

Table 3.5: Effect of β on the Sphere Shape Solutions

Note in Table (3.5) that for very small values of β, the camera rotation angle, the solution obtained is highly erroneous.

λ versus k_1

From the previous two variations, we find that we get best results for $\beta \approx 1.475$, and $k_3 \approx 0.7500$. We will use these values in the next case.

The effect of gray level quantization is very clear in Table (3.6), the results do not start to become acceptable until k_1 becomes 512. Note that this is performed with the best combination of β and k_3 obtained from above.

λ versus Image Size

The following data was obtained for $k_2 = 0.0, k_3 = 0.7500$
The size of the image was 64×64.
The rotation angle of the right camera $\beta = 1.475$ degrees.
Normalization is done about λ_4.

k_1	λ_0	λ_4	λ_7	λ_9
16.0	-7.171405	1.000000	0.502592	0.204058
32.0	-4.984379	1.000000	0.451841	0.268776
64.0	-17.031454	1.000000	0.736041	-0.374813
128.0	-17.369817	1.000000	0.810806	0.025142
256.0	-16.572301	1.000000	0.959069	0.718738
512.0	-16.161894	1.000000	0.972644	0.868655
1024.0	-15.845913	1.000000	0.990752	1.024426
2048.0	-15.993857	1.000000	0.995443	1.010252
4096.0	-16.010043	1.000000	0.997509	1.013835
8192.0	-16.006061	1.000000	0.998338	1.022081

Table 3.6: Effect of k_1 on the Sphere Shape Solutions

The following data was obtained for $k_1 = 4096.0$, $k_2 = 0.0, k_3 = 0.7500$
The rotation angle of the right camera $\beta = 1.475$ degrees.
Normalization is done about λ_4.

DIM	λ_0	λ_4	λ_7	λ_9
16	-15.991239	1.000000	0.870979	0.905877
32	-16.130688	1.000000	0.987538	1.011918
64	-16.010043	1.000000	0.997509	1.013835
128	-15.929016	1.000000	0.996361	1.017398

Table 3.7: Effect of Image Size on the Sphere Shape Solutions

Observations

The above tables give an idea of how the various parameters present in the problem affect the solution. Tables (3.4) and (3.5) present the variation in the results as we vary the angle of rotation β and the factor k_3. Although it is true that very small values of β give erroneous results, increasing β indefinitely does not *guarantee* good results. A very small value of β means that the change of the images as seen from the left camera or the right camera is very small, and may be lost in the quantization process. This, clearly, would result in errors in obtaining the image gradients, and thus errors in recovering the depth and reconstructing the structure.

By choosing the best possible combination of β and k_3 from Tables (3.4) and (3.5), the other parameters are studied as we see in Tables (3.6) and (3.7).

Chapter 4

CONCLUSIONS

In this thesis, the UOFF method for recovering surface structure has been used. We showed that all the terms necessary for the computations, are readily available from image data. We shall briefly discuss the major sources of errors encountered, and talk about the differences of this work from previous works.

4.1 Sources of Error

There are many sources of error that can affect the kind of computation that we're performing. To help us see the various sources of error, Fig. (4.1) below is produced. It shows a sample of the process followed in computing Q, the depth, for the Sphere shape, at a particular pixel.

The names of the variables shown in the figure are those used in the program, and they are very similar to the names used in in the theoretical work.

Quantization Errors

Fig. 4.1 shows what steps are taking place during the calculations done to get Q, at row #9, and column #28. The two small windows shown, are opened around the concerned pixel, for us to see the data that the program is using. The brightness invariance equation is as we've seen:

$$\frac{\partial g}{\partial x}\vec{u_s} + \frac{\partial g}{\partial y}\vec{v_s} + \frac{\partial g}{\partial \vec{s}} = 0$$

where from Eq. (2.24) and Eq. (2.25), u_s and v_s are:

$$u_s = (-\frac{U_s}{Z} - B_s + C_s y) - x(-\frac{W_s}{Z} - A_s y + B_s x)$$

and,

$$v_s = (-\frac{V_s}{Z} + A_s - C_s x) - y(-\frac{W_s}{Z} - A_s y + B_s x)$$

To test how close the *calculated* terms in the BIE are to zero, we use the Actual depth, variable `Az`, which is obtained from the image generating subroutines, and is present throughout the program.

Please note that `Az` is **not** used to solve anything; it is there for testing purposes, to enable the user to track the errors and see how good the depth recovering is. Using the correct u_s and v_s, we see that $\frac{\partial g}{\partial x}\vec{u_s} + \frac{\partial g}{\partial y}\vec{v_s} = 59.149503$.
And g_s = -78.5, a little too big, it should be -59.15 for the BIE to be exactly satisfied.

It was observed that, as β, or k_3, become smaller or larger, g_s becomes smaller or larger accordingly, which is not surprising. It is this *choice* of some parameters in the *simulation* part, that affects the results. The question of where to choose β and k_3, such that g_s is not too large, but yet not too small.

It was also observed, that as the dynamic range of the gray level function increases, g_s, (and g_x and g_y), tend to have less quantization error, and produce better results. Gray level quantization is one of the most penetrating errors, because almost all of the terms used in the recovery of the depth depend on the gray level values read from the image array. Namely, g_x, g_y, and g_s, all depend on the gray levels of the image array.

The recovered depth in Fig.(4.1) was $Z' = -0.837082$, or $Z = D + Z' = 99.16$, while the actual Z' was $Z'_a = -0.624162$; a difference of about 25%. Remember that the calculations necessary to compute the M matrices will need to use the depth information extensively, and therefor a small error my accumulate.

Fig.(4.2) shows the same calculations for a different pixel in which the depth recovered was more accurate.

The error here between the actual depth and the recovered depth is about 0.3%.

4.1.1 Loss of significance

Due to the large difference of magnitude between the values of X and Y, and the value of Z, some numerical error is inevitable. Q^{-1} is Z, and should be close to the value of D, namely 100. So Q is in the order of 0.01, and any variations in Q, takes place 3 digits after the decimal point. Eq. (2.26) calculates Q; but if there is a 10% error in Q, it translates to about 20% error in Z', because the perspective projection is a nonlinear transformation. In other words if the actual Q is 0.011, and the recovered Q is 0.0121 (10 % error), then the actual Z' is -9.091, and the recovered Z' is -17.35, 45% error.

4.1.2 Singularities

Due to the gray level quantization, and the sampling effect on x, and y, the dimensions on the screen, we may have some points in the image where one of the gradients becomes zero. This has shown to produce huge errors which could have a tremendous effect on the final output if not handled in proper manner. It is very difficult to predict this type of error, however a better way to deal with it is that if the value of the recovered z' is outside a certain range, i.e. too far from zero, that particular pixel is *excluded* from the computations in M and D. This method, however, if the range is not carefully chosen, may end up excluding too many pixels, and thus still affecting the results. In the experiments performed using the provided program, the percentage of the points that had to be excluded was always less than 0.1% for runs with good results.

4.1.3 Unbounded Surface

The Alpha shape surface used here is an *unbounded* (infinite) surface, i.e. not closed. This is one of the factors that affect the accuracy of the reconstruction, sinc the Algorithm does not see all the surface. That is why higher errors were experienced with the Alpha shape, than with the Sphere shape.

4.2 Accomplishments

The major accomplishments of this work can be summerized in the following:

- The Algorithm used starts with no knowledge at all about the image, and therefore is a practical and realistic approach.

- g_x, g_y, and, g_s are approximated using standard image processing techniques, while in [6], they were evaluated analytically from a known structure gray function distribution. In [6] it was done that way to prove that the method is accurate if the brightness gradients were not a source of error, since the method was still new; here, we extend that to show that it works with the gradients obtained with the usual processing techniques.

- The gray levels of the two images are *integers.* This also adds to the quantization errors, but makes the approach more realistic.

- The program developed during the course work of this thesis, is a powerful tool for future research in this field. It has been included in the thesis, with a clear documentation. It allows the user to investigate the effect of most, if not all, of the parameters, without too much change in the source code.

- With the reasonably good results obtained here, we show that the UOFF method for recovering depth, and structure, is a robust, and capable technique.

- To make the algorithm more automatic, a formula for obtaining K, the number of variables in the polynomial, as a function of the degree N, was developed. It is used at the beginning of the program to evaluate K from the sole knowledge of N.

4.3 Future Research

There are many interesting potential extensions to the work presented here, that utilize the UOFF method in the field of recovering structure and motion:

- Many of the errors encountered here were, in one way or another, due to the simulation of the two stereo images. Although the simulation is correct, the gray level generating function used has a potential for causing more than a few singularities. In other words, the gray level function is too sensitive to some parameters, such as β, k_2, and k_3. If we could reduce this sensitivity, there is a good case that many of the errors will vanish.

- To run the program, the user has had to provide N, the degree, for the program to test at. If N is unknown to start with, a possible approach is to start from $N = 1$ and keep increasing N by 1 and computing the corresponding $\lambda's$. When the coefficients corresponding to a certain layer, do not change with the increase of N, it might mean that the previous layer had the highest powers and we can stop there. Work can be done here to test this algorithm and/or come with better solutions to this problem.

- In our work we normalized about the X^2 term, $\lambda(4)$, for both the Sphere and the Alpha shape. In [6], normalization was done with respect to the constant term $\lambda(0)$. In both cases, we had to know *something* about the original structure of the surface to know which term we can normalize about, and which we cannot. This could be an interesting area of investigation, if we want our algorithm to be as independent and robust, as possible.

```
Limg[ 9][28] = 4879     Rimg[ 9][28] = 4801
        j -->                           jr -->
     |  26    27    28    29    30 ||  26    27    28    29    30
     | ----------------------------||------------------------------
i= 7 |  10    10    10    10    10 ||  10    10    10    10    10
i= 8 |  10    10    10    10    10 ||  10    10    10    10    10
i= 9 |  10    10 |4879| 4056  3401 ||  10    10 |4801| 3977  3323
i=10 |4627  4174  3769  3397  3047 || 4548  4095  3690  3319  2971
i=11 |4111  3794  3497  3214  2943 || 4032  3716  3419  3137  2867

As = 0.000000   Bs = -0.025744  Cs = 0.000000
Us = 2.574076   Vs = 0.000000   Ws = 0.033135
x=-0.006111      y=0.039286

gx=-347636.363636       gy=514618.181818        gs=-78.500

dg/dx * u_s :  -347636.3636 * -0.00015990   : 55.589114609 +
dg/dy * v_s :   514618.1818 *  0.00000691   : 3.5603892125 +
gs          :                               : -78.50000000 =
dg/dx*u_s + dg/dy*v_s + gs                  = -19.35049618

p=9031.428563   q=895582.810721 Q=0.010084      z = -0.837082
                                               Az = -0.624162
```

Figure 4.1: Run Sample 1

```
Limg[18][30] = 2775      Rimg[18][30] = 2701
 j -->                           jr -->
      | 28    29   30    31    32 ||  28    29    30    31    32
      | ---------------------------||-----------------------------
i =16 |3130  2961  2796  2633  2474 || 3054  2885  2721  2560  2402
i =17 |3102  2942  2784  2630  2478 || 3026  2866  2710  2556  2406
i =18 |3080  2926 |2775| 2627  2481 || 3004  2851 |2701| 2553  2409
i =19 |3061  2913  2767  2624  2483 || 2985  2838  2693  2551  2411
i =20 |3046  2902  2761  2622  2485 || 2970  2827  2687  2549  2413

As = 0.000000   Bs = -0.025744  Cs = 0.000000
Us = 2.574076   Vs = 0.000000   Ws = 0.033135
x=-0.002619      y=0.023571

gx=-84509.090909        gy=3054.545455  gs=-73.750000

dg/dx * u_s :   -84509.0909 * -0.00086994    : 73.518252911 +
dg/dy * v_s :   3054.545454 *  0.00000648    : 0.0198108222 +
gs          :                                : -73.75000000 =
dg/dx*u_s + dg/dy*v_s + gs         = -0.211936266

p=2249.338557   q=217542.567908 Q=0.010340      z =-3.285983
                                               Az =-3.276869
```

Figure 4.2: Run Sample 2

Bibliography

[1] B. K. P Horn and B. G. Scunck, "Determining optical flow," *Artificial Intelligence* 17 (1981) pp. 185-203

[2] B. Hayashi and S. Negahdaripour, "Direct motion stereo," *Proceedings SPIE vol 1260 Sensing and Reconstruction of Three-Dimensional Objects and Scenes,* pp. 78-85, Feb. 1990.

[3] S. Negahdaripour and B. P. Horn, "Direct passive navigation," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. PAMI-9, no. 1, pp. 168-176, Jan 1987

[4] B. K. P. Horn and E. J. Weldon Jr., "Direct Methods for Recovering Motion," *International Journal of Computer Vision,* 2, 51-76 (1988).

[5] C. Q. Shu and Y. Q. Shi, "On unified optical flow field," *Pattern Recognition,* (Accepted)

[6] C. Q. Shu, Y. Zhu, Y. Q. Shi, and, C.H.Lu, "Recovering surface structure characterized by an N-th Order polynomial equation," *IEEE Seventh Workshop on Multidimensional Signal Processing,* September 23-25, 1991, Lake Placid, NY. (Accepted)

Appendix A

Finding K

The following procedure is developed to compute K.

For a polynomial degree of N, there are $(N+1)$ *layers* that make up the entire polynomial. Each layer contains coefficients of a single order n; $n = 0, 1, \ldots, N$. At any specific layer, $\alpha_j + \beta_j + \gamma_j = n$. We can imagin $\alpha_j, \beta_j, \lambda_j$ as n stones that are thrown randomly on top of X, Y, and Z.

The question is, in how many different ways can the stones settle for each n ? If we imagin X, Y, and Z, as three boxes in which the falling stones will land, then there are two walls separating the stones. Note that the relative position of the stones is unimportant, while the relative position of the walls, and thus X, Y, Z, is important. If we let m equal the number of dimensions that we have, in our case, it is three: X, Y, Z, then we have the problem of combining n indistinguishable objects with $(m-1)$ distinguished objects.

This is similar to the Bose-Einstein problem, and the solution is that the number of combinations per layer is

$$\begin{pmatrix} (m+n-1)! \\ (m-1)! \end{pmatrix}$$

which if worked out yields

$$\frac{(m+n-1)!}{n!(m-1)!}$$

possible different combinations per layer. K, therefore, is the summation of all the possible $(N+1)$ layers, or:

$$K = \sum_{n=0}^{N} \frac{(m+n-1)!}{n!(m-1)!} \tag{A.1}$$

This formula is used at the beginning of the program for the computation of K knowing $m = 3$, and N.

Appendix B

The Program

```
/* The following program has been developed as a testing tool
 for the Unified Optical Flow Field approach, for reconstructing
the surface structure of a surface from two images taken
 of it by two cameras

The program is divided into three major parts:

1.  Building the images that the analysis will be
    conducted upon later
2.  Recovering the depth of the world point facing
    each pixel in the left image plane.
3.  Solving the optimization problem to recover the
       surface structure.

The last two parts do not use any of the information given to
 the first part. They deal only with the image pairs generated
in the first part.
The names of the variables used has been kept as close as possible
to those used in the thesis.

Global Variables:
N The order of the polynomial that we want to perform
the exp. for, (1, 2, 3, ...etc)
IDIM No. of rows used in the image screen
JDIM No. of columns used in the image screen
BKGND Gray level assigned to the pixels of the image array
that don't
contain a part of the surface
```

```
k1,k2 variables used to vary the gray generating fn. for creating
the images
Limg, Rimg  integer arrays of size IDIM*JDIM each, containing the
 Left and Right image arrays.
As,Bs,Cs Rotation rates of the right camera
Us,Vs,Ws Translation rates of the right camera
a, b, c alpha, beta, and gamma, used in the surface polynomial
K The number of coefficients in the polynomial
0.11*0.11 The size of image plane in real world coordinate units.
OFFSTx   x value we want left border of image plane to start at.
OFFSTy   y value we want top  border of image plane to start at.
Zb Distance "D" between center of surface and world coor. origin.
```

This program was designed to be relatively flexible, enabling its user to conduct a large number of different experiments without much changing the source code.

```
# begin{sf}
# include <stdio.h>
# include <math.h>
# define N 3
# define IDIM 128
# define JDIM 128
# define BKGND 10
# define k1 1000.0
# define k2 0.0
# define OFFSTx (-0.055)
# define OFFSTy ( 0.065)
# define PI (4.0*atan(1.0))
# define ERR 1.0e-5
# define TINY 2.0e-7
# define Zb 100.0
# define M_SAVING_FORMAT "%d\t%d\t%le\n" /* Format for saving M */
# define D_SAVING_FORMAT "%d\t%le\n"  /* Format for saving D */
```

THE SUBROUTINES

```
void  testfor(); /* Performs complete procedure for the
   given variables*/
```

```
void   drawA(); /* Draws the Alpha  image */
double Aroots();
void   drawS();         /* Draws the Sphere image */
double Sroots();
double dmax();
int    fctrl();
void   imgmake();  /* Saves the image drawn in a file */

void   stdrabc(); /* Loads a,b,c with standard setting */
short  int getK(); /* Finds K from N */
void   ABCUVW(); /* Loads Translational and rotational vars. */
void   knowtask();
void   imgread();    /* Reads the saved image files */
void   prepare();
double getEx();
double getEy();
double getEs();
double getp();
double getq();
void   getQ();  /* Computes Q */

void   compM(); /* computes M */
void   compD(); /* computes D */
void   Mxsave();
void   Vcsave();
void   Lamsave();
void   Lamplot();
void   Mxload();
void   Vcload();
void   Gauss();
void   prepMD();
void   intrprt();

short int Limg[IDIM][JDIM], Rimg[IDIM][JDIM];
double As,Bs,Cs,Us,Vs,Ws,DLS;
short int a[20], b[20], c[20], Norm;
short int K;
double AQ[IDIM][JDIM];
double k3;

main()
/* main prog starts here */
```

```
{
int i;
double bet;

K = getK(N);
stdrabc();


/* The following do-while statements command the performance of the
 experiment for a variaty of bet and k3, as seen. Any variables of
 the testfor routine can be varied depending on the user's intension.
Here we are testing for an Alpha image ("A"), and normalizing
about lambda 4 */

bet = 0.1;
while ((bet>=0.0) && (bet<1.5))
    {
    k3 = 0.5;
    while ((k3>=0.0) && (k3<4.0))
{
testfor("A", "TEST_A.128",  2, 1, bet,   4);
k3 += 0.125;
}
    bet += 0.125;
    }
}
/* main prog ends here */
/* ---------------------------------------------------- */
/* The following subroutine performs one experiment for
   the given set of variables.

INPUTS:

Surname "A" for the Alpha image, "S" for the Sphere image,
extendible.
Lname Name of file to save result in.
ndx used in calculating x and y grad.: 1 or 2 only,
see getEx, getEy
nds used in calculating s grad.  : 1 or 2 only,
see getEs.
Beta Anlge of rotation of right camera, Degrees.
norm element in polynomial to normalize about.
```

```
OUTPUTS:
    * Save the M and D matrices  (Optional)
    * Save the result of Lambda in file Lname (Optional)
    *  Save selected Lammbdas for plots (for multp. runs)
*/

void testfor(Surname,Lname,ndx,nds,Beta,norm)
char Surname[2],Lname[20];
int ndx, nds, norm;
double Beta;
{
FILE *fp;
short int i, j, I, J, Lin[IDIM][JDIM], ALin[IDIM][JDIM];
double X, Y;
double M[22][22], D[22], L[22], sen, sad;
double Q[IDIM][JDIM];
char Mname[22], Dname[22];
char Limgnm[22], Rimgnm[22];


Beta = Beta*PI/180.0;
Norm = norm;
knowtask (Lname,Mname,Dname,Limgnm,Rimgnm);
ABCUVW(Beta,nds);

     if (Surname[0] == 'A') drawA(Beta,AQ,ALin);
else if (Surname[0] == 'S') drawS(Beta,AQ,ALin);
else
{
printf("Unknown drawing request\n");
exit();
}

prepare(Lin);
getQ(Lin,Beta,ndx,Q);

compM(Q,Lin,M,Norm);
compD(Q,Lin,D,Norm);
prepMD(M,D,Norm);
Gauss(M,D,(K-1),L);
intrprt(L,Norm);
```

```
Lamplot ((K-1),L,Norm,Beta);

/* Lamsave (Lname,(K-1),L,Norm,Beta,ndx,nds);
Lamsave is best used if a single experiment is run, because it
would consume a lot of memory otherwise and be difficult
to comprehend */

}
/* ---------------------------------------------------------- */
/* getQ is the subroutine used to recover the depth
INPUTS:
in int flagame size as img array, signals pixel is in(1) or out(0).
Beta Angle of rotation of left camera
ndx used in getEx, and getEy subroutines

OUTPUTS:
in modified to exclude some points where recovered depth
is not reasonable
Q (1/Z), the recovered depth.*/

void getQ(in,Beta, ndx,Q)
short int in[IDIM][JDIM];
double Beta;
double Q[IDIM][JDIM];
int ndx;
{
int i,j, count, outcnt, negcnt;
double Ex, Ey, Es , x, y, z, Az;
double p,q, Qav, Qsum;
double u_s, v_s;

Qsum=0.0;
count=0;
outcnt=0;
negcnt=0;

for (i=0; i<IDIM; i++)
    {

    for (j=0; j<JDIM; j++)
Q[i][j] = 0.00001;
    }
```

```
for (i=0; i<IDIM; i++)
    {
    for (j=0; j<JDIM; j++)
{
if ( i==(IDIM-1) || j==(JDIM-1) ) in[i][j] = 0;

if (in[i][j] == 0)  Q[i][j]=0.00001;
else if (in[i][j] == 1)
    {
    count += 1;

    Ex       = getEx(i,j,ndx);
    Ey       = getEy(i,j,ndx);
    Es       = getEs(i,j);
    x        = ( 0.11*(double)j)/(JDIM -1) + OFFSTx ;
            y        = (-0.11*(double)i)/(IDIM -1) + OFFSTy ;

    /*
    u_s      = (-Us*AQ[i][j] -Bs) - x*(-Ws*AQ[i][j] + Bs*x);
    v_s      = y*(Ws*AQ[i][j] - Bs*x);
    Es       = -(Ex*u_s + Ey*v_s);
    See the brightness invariance equation,
    These are used to test intermediate states of
    the solution using Actual depth. They are NOT used
    in the reconstruction. If these statements are
    included in the program you should get a PERFECT
    solution (tested) */

    p        = getp(x,y,Ex,Ey,Es);
    q        = getq(x,y,Ex,Ey,Es);
    Q[i][j] = p/q ;

            z        = 1.0/Q[i][j]  - Zb;

    if (fabs(z)> 18.0  ||  Q[i][j]<(1.0/Zb))
    {
    /* if solution is too big, exclude it */
    in[i][j] = 0;
    outcnt += 1;
    }
    if (Q[i][j] < 0.0)
    {
    in[i][j] = 0; /*if Q<0 exclude it */
    negcnt  += 1;
    }
    }
}
    }
}
/* -------------------------------------------------- */
/* This subroutine uses the filename chosen in 'testfor'
   and generates similar names to save M and D in them. */
void knowtask(Lname,Mname,Dname,Limgnm,Rimgnm)
char Lname[20], Mname[20], Dname[20];
char Limgnm[20], Rimgnm[20];
{
int i,size;

for(i=0; i<15; i++)
    {
    Mname[i] = Lname[i];
    Dname[i] = Lname[i];
    }

Mname[ 0] = 'M';
Mname[ 2] = 'm';
Mname[ 7] = 'M';
Dname[ 0] = 'D';
Dname[ 2] = 'd';
Dname[ 7] = 'D';

Limgnm[ 0] = 'i';
Limgnm[ 1] = 'm';
Limgnm[ 2] = 'a';
Limgnm[ 3] = 'g';
Limgnm[ 4] = 'e';
Limgnm[ 5] = 's';
Limgnm[ 6] = '/';
Limgnm[ 7] = 'i';
Limgnm[ 8] = 's';
Limgnm[ 9] = Lname[4];
Limgnm[10] = Lname[5];
Limgnm[11] = '/';
```

```
Limgnm[12] = 'L';
Limgnm[13] = Lname[4];
Limgnm[14] = Lname[5];
Limgnm[15] = '.';
Limgnm[16] = Lname[11];
Limgnm[17] = Lname[12];

for(i=0; i<=17; i++)
    {
    if      (i != 12) Rimgnm[i] = Limgnm[i];
    else if (i == 12) Rimgnm[i] = 'R';
    }
}
/* ---------------------------------------------------------- */
/* This subroutine prepares the 'in' flag from the image data */
void prepare(Lin)
short int Lin[IDIM][JDIM];
{
int i,j;

for (i=0; i<IDIM; i++)
    {
    for (j=0; j<JDIM; j++)
{
if (Limg[i][j] == BKGND) Lin[i][j]=0;
else  Lin[i][j] = 1;
}
    }
}
/* --------------------------------------------------------- */
/* This subroutine computes the M matrix
INPUTS:
Q Depth information
in in flag
norm
*/
void compM(Q,in,M,norm)
double Q[IDIM][JDIM];
short int in[IDIM][JDIM];
double  M[22][22];
short int norm;
{
```

```
double powXm, powYm, powQm;
double   MX ,   MY ,   MQ, z, x, y, X,Y;
short int i,j,I,J;

for (I=0; I<=(K-1) ; I++)
    for (J=0; J<=(K-1) ; J++) M[I][J] = 0.0;

for (I=0; I<=(K-1); I++)
    {
    for (J=0; J<=(K-1); J++)
      {
if((I != norm) && (J !=  norm))
{
powXm = (double)(a[J] + a[I]);
powYm = (double)(b[J] + b[I]);
powQm = (double)(c[J] + c[I]);

for (i=0; i<IDIM; i++)
         {
    for (j=0; j<JDIM; j++)
{
z = (1.0/Q[i][j] - Zb);
if (in[i][j] == 1)
    {
    x = ( 0.11*(double)j)/(JDIM -1) + OFFSTx ;
    y = (-0.11*(double)i)/(IDIM -1) + OFFSTy ;
    X = x/Q[i][j];
    Y = y/Q[i][j];

    if (X == 0.0) MX=0.0;
    else MX = pow(X, powXm);

    if (Y == 0.0) MY=0.0;
    else MY = pow(Y, powYm);

    if (z == 0.0)  MQ=0.0;
    else if (powQm == 0.0)  MQ=1.0;
    else MQ = pow(z , powQm);
    }
else if (in[i][j] == 0)
    MX=0.0;
```

```
M[I][J] = M[I][J] + MX*MY*MQ ;
}
    }
}
      }
    }
}
/* ---------------------------------------------------- */
/* This subroutine computes the D vector */
void compD(Q,in,D,norm)
double D[22];
double Q[IDIM][JDIM];
short int in[IDIM][JDIM];
short int norm;
{
double powXd, powYd, powQd;
double   DX ,   DY ,   DQ , z , x, y , X, Y;
short int i,j,I;

for (I=0; I<=(K-1); I++)  D[I] = 0.0;

for (I=0; I<=(K-1); I++)
  {
    if (I != norm)
    {
    powXd = (double)(a[I] + a[norm]);
    powYd = (double)(b[I] + b[norm]);
    powQd = (double)(c[I] + c[norm]);

    for (i=0; i<IDIM; i++)
    {
for (j=0; j<JDIM; j++)
    {
    z = ((1.0/Q[i][j]) - Zb);
    if (in[i][j] == 1)
{
x  = ( 0.11*(double)j)/(JDIM -1) + OFFSTx ;
y  = (-0.11*(double)i)/(IDIM -1) + OFFSTy ;
    X  = x/Q[i][j];
    Y  = y/Q[i][j];

 if (X == 0.0) DX=0.0;
```

```
else DX = pow(X, powXd);

if (Y == 0.0) DY=0.0;
else DY = pow(Y, powYd);

if (z == 0.0)  DQ=0.0;
else if (powQd == 0.0)  DQ=1.0;
else DQ = pow(z ,powQd);
}

    else if (in[i][j] == 0)
DX = 0.0;

    D[I] = D[I] - DX*DY*DQ ;
    }
}
    }
  }
}
/*-----------------------------------------------------------*/
/* This subroutine is used to save the M matrix */
void Mxsave(fname, I, J,M,norm)
char *fname;
int I,J, norm;
double M[20][20];

{
FILE *fp;
int i,j;

    if ((fp=fopen(fname, "w")) == NULL)
{ /* if there is error in opening file */
printf("Error in opening file %10s \n",fname);
exit();
}
    else
{
for(i=0; i<= I; i++)
    {
    for (j=0; j<= I; j++)
if((i!=norm)&&(j!=norm))
    fprintf(fp,M_SAVING_FORMAT,i,j,M[i][j]);
    }
}
fclose(fp);
}
/*-----------------------------------------------------------*/
/* This subroutine is used to save the D vector */
void Vcsave(fname, I,V,norm)
char *fname;
int I,norm;
double V[20];

{
FILE *fp;
int i;

    if ((fp=fopen(fname, "w")) == NULL)
{ /* if there is error in opening file */
printf("Error in opening file %10s \n",fname);
exit();
}
    else
{
for(i=0; i<= I; i++)
    if(i!=norm)
fprintf(fp,D_SAVING_FORMAT,i,V[i]);
}
fclose(fp);
}
/* ---------------------------------------------------------- */
double getp(x, y, Ex, Ey, Es)
double x,y,Ex,Ey,Es;
{
double p;
p = (double)((-As*y+Bs*x)*(x*Ex+y*Ey) - Ex*(-Bs+Cs*y)
    - Ey*(-Cs*x+As) - Es);
return p;
}
/* ---------------------------------------------------------- */
double getq(x, y, Ex, Ey, Es)
double x,y,Ex,Ey,Es;
{
double q;
```

```
q = (double)((x*Ws*Ex) + (y*Ws*Ey) - (Us*Ex) - (Vs*Ey));
return q;
}
/* ---------------------------------------------------------- */
double getEx(i,j,ndx)
short int i,j,ndx;
{
double Ex,gx,dx;

gx = (double)(Limg[i][j+1]-Limg[i][j]+Limg[i+1][j+1]-Limg[i+1][j]
     +  Rimg[i][j+1]-Rimg[i][j]+Rimg[i+1][j+1]-Rimg[i+1][j] );

if (gx == 0.00) gx=1.0e-9;

if      (ndx == 1) dx = 1.0;
else if(ndx == 2) dx = (double)(0.11/JDIM);

Ex = 0.25*gx/dx;

return Ex;
}
/* ------------------------------------------------------- */
double getEy(i,j,ndx)
short int i,j,ndx;
{
double Ey,gy,dy;

gy = (double)(Limg[i+1][j]-Limg[i][j]+Limg[i+1][j+1]-Limg[i][j+1]
     +  Rimg[i+1][j]-Rimg[i][j]+Rimg[i+1][j+1]-Rimg[i][j+1] );

if (gy == 0.00) gy=1.0e-9;

if      (ndx == 1) dy = -1.0;
else if(ndx == 2) dy = (double)(-0.11/IDIM);

Ey = 0.25*gy/dy;

return Ey;
}
/* ------------------------------------------------------- */
double getEs(i,j)
short int i,j;
```

```
{
double gs,Es;


gs = (double)(Rimg[i][j]-Limg[i][j] + Rimg[i+1][j]-Limg[i+1][j]
     +  Rimg[i][j+1]-Limg[i][j+1] + Rimg[i+1][j+1]-Limg[i+1][j+1]);

Es = 0.25*gs/DLS;

return Es;
}
/* -------------------------------------------------------- */
/* This subroutine computes K */
short int getK(Num)
int Num;
{
short int m=3;
int n,k;
k=0;

for (n=0; n<=Num; n++)
    k = k + (fctrl(m+n-1))/(fctrl(n)*fctrl(m-1));
return k;
}
/* ---------------------------------------------------------- */
/* calculates the factorial of input n */
int fctrl(n)
short int n;
{
int f,i;
f=1;

if (n >= 1)
    {
    for (i=1; i<=n; i++) f=f*i;

    return (f);
    }
else return 1;
}
/* ---------------------------------------------------------- */
/* This subroutine loads a,b, and c, with the standard setting
```

```
   assumed in the reconstruction */
void stdrabc()
{
short z;
z=200;
a[0]  = 0;        b[0]  = 0;        c[0]  = 0;
a[1]  = 1;        b[1]  = 0;        c[1]  = 0;
a[2]  = 0;        b[2]  = 1;        c[2]  = 0;
a[3]  = 0;        b[3]  = 0;        c[3]  = 1;
a[4]  = 2;        b[4]  = 0;        c[4]  = 0;
a[5]  = 1;        b[5]  = 1;        c[5]  = 0;
a[6]  = 1;        b[6]  = 0;        c[6]  = 1;
a[7]  = 0;        b[7]  = 2;        c[7]  = 0;
a[8]  = 0;        b[8]  = 1;        c[8]  = 1;
a[9]  = 0;        b[9]  = 0;        c[9]  = 2;
a[10] = 3;        b[10] = 0;        c[10] = 0;
a[11] = 2;        b[11] = 1;        c[11] = 0;
a[12] = 2;        b[12] = 0;        c[12] = 1;
a[13] = 1;        b[13] = 2;        c[13] = 0;
a[14] = 1;        b[14] = 1;        c[14] = 1;
a[15] = 1;        b[15] = 0;        c[15] = 2;
a[16] = 0;        b[16] = 3;        c[16] = 0;
a[17] = 0;        b[17] = 2;        c[17] = 1;
a[18] = 0;        b[18] = 1;        c[18] = 2;
a[19] = 0;        b[19] = 0;        c[19] = 3;

}
/* -------------------------------------------------- */
/* This subroutine is used for saving the solutions of
   a multiple run experiment in a plot ready format
NOTE THAT YOU NEED TO CHANGE THE FILE NAMES FROM HERE
AND TO MAKE SURE THAT ANY OLD VERSION OF THE SAME FILE
NAME IS ERASED BEFORE THE NEW RUN,
otherwise it would just append at the end of the old file
*/
void Lamplot(I,L,norm,BB)
int I,norm;
double L[22],BB;

{
FILE *fp1, *fp2;
int i;
```

```
L[norm] = 1.0;

if ((fp1=fopen("Plots/Anz", "a+")) == NULL)
    { /* if there is error in opening file */
    printf("Error in opening file %10s \n","Plots/Anz");
    exit();
    }
else
    fprintf(fp1,"Beta = %3.4f\tk3 = %1.4f\t\t
       %2.6f\t%2.6f\t%2.6f\t%2.6f\t\n",
    (BB*180.0/PI),k3,L[4],L[7],L[9],L[16]);
fclose(fp1);


if ((fp2=fopen("Plots/Az", "a+")) == NULL)
    { /* if there is error in opening file */
    printf("Error in opnning file %10s \n","Plots/Az");
    exit();
    }
else
    {
    fprintf(fp2,"Be=%3.3f  k3=%1.4f\t",(BB*180.0/PI),k3);
    for(i=0; i<=13; i++)
{
if(i!=4 && i!=7 && i!=9 && i!=16)
        fprintf(fp2,"%2.3f  ",L[i]);
}
    fprintf(fp2,"\n");
    }
fclose(fp2);

}
/* ------------------------------------------------------ */
/* This subroutine Reads the saved image */
void imgread(fname, fp, intimg)
short intimg[IDIM][JDIM];
FILE *fp;
char *fname;
{
    short i,j,c;
    if ((fp=fopen(fname, "r")) == NULL)
```

```
{ /* if there is error in opening file */
printf("Error in opening file %10s \n",fname);
exit();
}
    else
{
for(i= 0; i< IDIM; i++)
    {
    for (j=0; j< JDIM; j++)
{
intimg[i][j] = getc(fp);
}
    }
}
        fclose(fp);
}
/*--------------------------------------------------------*/
/* makes two image files from the two integer arrays
Limg and Rimg */
void imgmake(fname, fp, intimg)
short intimg[IDIM+1][JDIM+1];
FILE *fp;
char *fname;

{
    int i,j;

    if ((fp=fopen(fname, "w")) == NULL)
{ /* if there is error in opening file */
printf("Error in opening file %10s \n",fname);
exit();
}
    else
{
for(i=0; i< IDIM; i++)
    {
    for (j=0; j< JDIM; j++)
{
putc(intimg[i][j], fp);
}
    }
}
```

```
fclose(fp);
}
/* ---------------------------------------------------- */
void ABCUVW(Beta,nds)
double Beta;
int nds;
{
double xht, zht;
xht = (Zb*sin(Beta));
zht = (Zb*(1.0 - cos(Beta)));
if      (nds == 1) DLS = 1.0;
else if (nds == 2) DLS = sqrt(xht*xht + zht*zht + Zb*Zb*Beta*Beta);
else    ("Check nds !!!!\n");

As = 0.0;
Bs = (-Beta)/DLS;
Cs = 0.0;
Us = (Zb*sin(Beta))/DLS;
Vs = 0.0;
Ws = (Zb*(1.0 - cos(Beta)))/DLS;

}
/* ---------------------------------------------------- */
void intrprt(L, norm)
double L[22];
int norm;
{
int i,I;
double l[22];

for(i=1; i<=(K-1); i++)
    l[i] = L[i];

for (I=0; I<=(K-1); I++)
    {
    if (I<norm) L[I] = l[I+1];
    else if (I>norm) L[I] = l[I];
    }
}
/* ---------------------------------------------------- */
void prepMD(M, D, norm)
double M[22][22], D[22];
```

```
int norm;
{
double m[22][22], d[22];
int i,j,I,J;

for (I=0; I<=(K-1); I++)
    {
    for (J=0; J<=(K-1); J++)
if ((I!=norm) && (J!=norm)) m[I][J] = M[I][J];

    if (I != norm)  d[I] = D[I];
    }

for (I=1; I<=(K-1); I++)
    {
    for (J=1; J<=(K-1); J++)
{
if (I<=norm) i=I-1;
else if (I>norm) i=I;

if (J<=norm) j=J-1;
else if (J>norm) j=J;

M[I][J] = m[i][j];
}
    D[I] = d[i];
    }
}
/* ----------------------------------------------------- */
/* This subroutine solves the system of linear equations ML=D */
void Gauss(M,D,n,x)
double M[22][22], D[22], x[22];
int n;
{
int i,j,k, lk, l[22];
double A[22][22],B[22],S[22],smax,rmax,r,xmult,sum ;

for (i=1; i<=n; i++)
    {
    for (j=1; j<=n; j++)
A[i][j] = M[i][j];
    B[i] = D[i];
    }

for (i=1; i<=n; i++)
    {
    l[i] = i;
    smax = 0.0;
    for (j=1; j<=n; j++)
smax = dmax(smax,(double)(fabs(A[i][j])));
    S[i] = smax;
    }

for (k=1; k<=(n-1); k++)
    {
    rmax = 0.0;
    for (i=k; i<=n; i++)
{
r = (double)(fabs(A[(l[i])][k] / S[(l[i])]));
if (r > rmax)
    {
    j = i;
    rmax = r;
    }
}
lk = l[j];
l[j] = l[k];
l[k] = lk;
for (i=(k+1); i<=n; i++)
    {
    xmult = A[l[i]][k] / A[lk][k];
    for (j=(k+1); j<=n; j++)
A[l[i]][j] = A[l[i]][j] - xmult*A[lk][j];
    A[l[i]][k] = xmult;
    }
    }

for (k=1; k<=(n-1); k++)
    {
    for (i=(k+1); i<=n; i++)
B[l[i]] = B[l[i]] - A[l[i]][k]*B[l[k]];
    }

x[n] = B[l[n]] / A[l[n]][n];
```

```
for (i=(n-1); i>=1; i--)
    {
    sum = B[l[i]];
    for (j=(i+1); j<=n; j++)
sum = sum - A[l[i]][j]*x[j];
    x[i] = sum/A[l[i]][i];

    }
}
/* ------------------------------------------------------- */
double dmax(a,b)
double a,b;
{
double x;
x = a;
if (b>x) x=b;
return x;
}
/* ------------------------------------------------------- */
/* This subroutine saves the solution of a single run,
   along with many of the variables that one run from another */
void Lamsave(fname, I,V,norm,BB,ndx,nds)
char *fname;
int I,norm,ndx,nds;
double V[22],BB;

{
FILE *fp, *fp2, *fp3;
int i;

    if ((fp=fopen(fname, "a+")) == NULL)
{ /* if there is error in opening file */
printf("Error in opening file %10s \n",fname);
exit();
}
    else
{
fprintf(fp,"\n\n----------------------------\n");
fprintf(fp,"\n \t IDIM =\t%d\n",IDIM);
fprintf(fp,"\nThis result is for Zb=%g\n",Zb);
fprintf(fp,"And for Beta=%g\n\n",BB*180.0/PI);
fprintf(fp,"As = %f\tBs = %f\tCs = %f\nUs = %f\t
    Vs = %f\tWs = %f\n",As,Bs,Cs,Us,Vs,Ws);
fprintf(fp,"k1 = %f\tk2 = %f\tk3 = %f\n",k1,k2,k3);
fprintf(fp,"ndx=%d    nds=%d\n",ndx,nds);
fprintf(fp,"%30s",fname);
fprintf(fp,"\n\n");
for(i=0; i<= I; i++)
    if (i != norm)
fprintf(fp,"Lambda[%2d] = \t %1.12f\n",i,V[i]);
    else if (i == norm)
fprintf(fp,"Lambda[%2d] = \t %1.12f\n",i,1.00);
}
fclose(fp);
}
/* ------------------------------------------------------------ */
void Mxload(fname,Surname, I, J, M, norm)
char *fname;
int I,J,norm;
double M[22][2];
{
FILE *fp;
short i,j,c;
int l,m;
    if ((fp=fopen(fname, "r")) == NULL)
{ /* if there is error in opening file */
printf("Error in opening file %10s \n",fname);
exit();
}
    else
{
for(i= 0; i<= I; i++)
    {
    for (j=0; j<= J; j++)
{
if((i!=norm) && (j!=norm))
    {
    c = fscanf(fp, M_SAVING_FORMAT,&l,&m,&M[i][j]);
    if(c==EOF)printf("Error in loading file %9s",fname);
/*printf("Mxload: \tM[%2d][%2d] = %g \n",i,j,M[i][j]);*/

    }
}
```

```
    }
}
        fclose(fp);
}
/* -------------------------------------------------------- */
void Vcload(fname, I, V,norm)
char *fname;
int I,norm;
double V[22];
{
FILE *fp;
short i,c,l;
    if ((fp=fopen(fname, "r")) == NULL)
{ /* if there is error in opening file */
printf("Error in opening file %10s \n",fname);
exit();
}
    else
{
for(i= 0; i<= I; i++)
    {
    if(i != norm)
{
c = fscanf(fp, D_SAVING_FORMAT,&l,&V[i]);
if (c == EOF) printf("Error in loading file %9s",fname);
/*printf("Vcload: \tD[%2d] = %g \n",i,V[i]);*/
}
    }
}
        fclose(fp);
}
/* ------------------------------------------------------- */
/* This subroutine 'draws' the Alpha image and puts it in
   the arrays Limg and Rimg. It also returns the ACTUAL Q and "in"
   matrices to be used for testing by the user,
   NOT to be used in the reconstruction */

void drawA(Beta,Q,in)
double Beta;
double  Q[IDIM][JDIM];
short int in[IDIM][JDIM];
{
```

```
double X,Y,z,Z,x,y,XL, YL;
int i,j,gl,gr,soln;
double r,th,phi,Y2,Dr,Dl;
double Xi,Zet;
FILE *fp;
char *fname;

 for (i=0; i<IDIM; i++)
    {
    for (j=0; j<JDIM; j++)
{
Limg[i][j] = BKGND;
Rimg[i][j] = BKGND;
}
    }

 for (i=0; i<IDIM; i++)
    {
    for (j=0; j<JDIM; j++)
{
x = ( 0.11*(double)j)/(JDIM -1) + OFFSTx ;
y = (-0.11*(double)i)/(IDIM -1) + OFFSTy ;

Z = Aroots(x,y);

if (fabs(Z - Zb) <=80.0) soln=1;
else soln=0;

if(soln == 1)
    {
    XL = x*Z;
    YL = y*Z;
    z  = Z - Zb;
    r  = sqrt(XL*XL + z*z);
    th = atan( r/(YL+TINY));
    phi= atan(z/(XL+TINY));
if(XL<0.0) phi = phi+PI;
if(XL>=0.0 && z<0.0) phi = phi+2.0*PI;

    gl =(int)(k1*cos(k2*th)*sin(k3* phi       )+k1+0.5)+25;
    gr =(int)(k1*cos(k2*th)*sin(k3*(phi+Beta))+k1+0.5)+25;
    Limg[i][j] = (short)(gl);
    Rimg[i][j] = (short)(gr);
    Q[i][j]    = 1/Z;
    in[i][j]   = 1;
    }
else if (soln == 0)
    {
    gl         = BKGND;
    gr         = BKGND;
    Limg[i][j] = (short)gl;
    Rimg[i][j] = (short)gr;
    Q[i][j]    = 1/Z;
    in[i][j]   = 0;
    }
}
    }
}
/*-------------------------------------------------------*/
/* Given the position of the pixel on the screen,
   this subroutine calculates the Actual depth for the
   Alpha shape image */

double Aroots(l,m)
double l,m;
{
double a,b,c,d;
double fx,dfx,x,start;
int i;
double Z;

  /*  printf("x y %.6e\t%.6e\n",l,m); */
if (fabs(m)<TINY)
    {
    Z = 100000.0;
    return Z;
    }

a = m*m*m;
b = l*l - 5.0*m*m + 1.0;
c = -2.0*Zb;
d = Zb*Zb;
```

```
 b = b/a;
 c = c/a;
 d = d/a;
 a = a/a;
     /* printf("a b c d \t%.6e\t%.6e\t%.6e\t%.6e \n",a,b,c,d); */

 start = Zb - 5.0;

 x=start;
 fx = x*x*x + b*x*x + c*x + d ;

 for (i=0; i<= 12; i++)
     {
     dfx = 3.0*x*x + 2.0*b*x + c ;
     x = x - fx/dfx;
     fx = x*x*x + b*x*x + c*x + d ;
     }

/*  printf("i z fz \t%d\t%.10e\t%g \n",i,x,fx); */
    if ((fx > ERR) || (-1.0*fx > ERR))
{
Z=100000.0;
     /*   printf("No solution at this x,y \n"); */
return Z;
}

    else
{
        Z = x;
     /*   printf("Solution found at current x,y \n"); */
     /*   printf("Z \t%.5g\n",Z); */
return Z;
}

}
/*------------------------------------------------------------*/
/* This subroutine 'draws' the Sphere image planes and puts them
   in Limg and Rimg. It also returns the ACTUAL Q and in matrices
   to be used for testing by the user,
   NOT to be used in the reconstruction */

void drawS(Beta,Q,in)
```

```
double Q[IDIM][JDIM];
short int in[IDIM][JDIM];
double Beta;
{
double z,Z,x,y,xr,yr, X,Y, XL, YL;
int i,j,gl,gr,soln;
double r,th,phi;
FILE *fp;
char *fname;

 for (i=0; i<IDIM; i++)
    {
    for (j=0; j<JDIM; j++)
{
Limg[i][j] = BKGND;
Rimg[i][j] = BKGND;
}
    }

 for (i=0; i<IDIM; i++)
    {
    for (j=0; j<JDIM; j++)
{
x = ( 0.11*(double)j)/(JDIM -1) + OFFSTx ;
y = (-0.11*(double)i)/(IDIM -1) + OFFSTy ;

Z = Sroots(x,y);

if (fabs(Z - Zb) <=80.0) soln=1;
else soln=0;

if(soln == 1)
    {
    XL = x*Z;
    YL = y*Z;
    z  = Z - Zb;
    r  = sqrt(XL *XL  + z*z);
    th = atan( r/(YL+TINY));
    phi= atan( z/(XL+TINY));
if(XL<0.0) phi = phi+PI;
if(XL>=0.0 && z<0.0) phi = phi+2.0*PI;
```

```
    gl =(int)(k1*cos(k2*th)*sin(k3* phi      )+k1+0.5)+25;
    gr =(int)(k1*cos(k2*th)*sin(k3*(phi+Beta))+k1+0.5)+25;

    Limg[i][j] = (short)(gl);
    Rimg[i][j] = (short)(gr);
            Q[i][j]     = 1/Z;
    in[i][j]    = 1;
    }
else if (soln == 0)
    {
    XL  = 0.0;
    YL  = 0.0;
    gl  = BKGND;
    gr  = BKGND;
    Limg[i][j] = (short)(gl);
    Rimg[i][j] = (short)(gr);
            Q[i][j] = 1/Z;
    in[i][j]    = 0;
    }
}
    }

/*
for (i=0; i<IDIM; i++)
    {
    for (j=5; j<20; j++) printf("%4d",Limg[i][j]);
    printf("\t");
    for (j=5; j<20; j++) printf("%4d",Rimg[i][j]);
    printf("\n");
    }
*/

}
/*-----------------------------------------------------------*/
double Sroots(l,m)
double l,m;
{
double a,b,c,d;
int i;
double Z;

  /*  printf("x y %.6e\t%.6e\n",l,m); */
```

```
a = l*l + m*m + 1.0;
b = -2.0*Zb;
c = Zb*Zb - 16.0;

d = b*b - 4.0*a*c;

if (d < 0.0)
    {
    Z = 100000.0;
    }
else if (d>=0.0)
    {
    Z = (-b - sqrt(d))/(2.0*a);
    }

/* printf("x=%1.4f   y=%1.4f \t\tZ=%f\n",l,m,Z); */
return Z;
}
/*--------------------------------------------------------*/
```