2006

Close range three-dimensional position sensing using stereo matching with Hopfield neural networks.

Houman Rastgar
University of Windsor

CLOSE RANGE 3D POSITION SENSING USING STEREO MATCHING WITH HOPFIELD NEURAL NETWORKS

By

Houman Rastgar

A Thesis
Submitted to the Faculty of Graduate Studies and Research through the Department of Electrical and Computer Engineering in Partial Fulfillment of the Requirements for the Degree of Master of Applied Science at the

University of Windsor

Windsor, Ontario, Canada
2006

Your file Votre référence
ISBN: 978-0-494-17107-3
Our file Notre référence
ISBN: 978-0-494-17107-3

1043458

ABSTRACT

In recent years Vision Systems have found their ways into many real-world applications. This includes such fields as surveillance and tracking, computer graphics and various factory settings such as assembly line inspection and object manipulation. The application of Computer Vision techniques to factory automation, Machine Vision, is a growing field. However in most Machine Vision systems an algorithm is needed to infer 3D information regarding the objects in the field of view. Such a task can be accomplished using a Stereo Vision algorithm.

In this thesis a new Machine Vision Algorithm for Close-Range Position Sensing is presented where a Hopfield Neural Network is used for the Stereo Matching stage: stereo Matching is formulated as an energy minimization task which is accomplished using the Hopfield Neural Networks. Various other important aspects of this Vision System are discussed including camera calibration and objects localization.

ACKNOWLEDGMENTS

I would like to give my sincere thanks to my supervisor, Dr. M. A. Sid-Ahmed. His encouragement and innovative ideas enabled me to successfully complete this thesis. I wish to thank Dr. M. Ahmadi for his comments and directions. I like to express my sincere appreciation to Dr. J. Chen for her recommendations and sincere support.

TABLE OF CONTENTS

LIST OF FIGURES

Number *Page*

LIST OF TABLES

Chapter 1: Introduction

1.1 Problem Statement

The aim of this thesis is to develop a Vision System for a Close-Range Position sensing application. In other words a "bin-picking" robotic vision system is to be developed. The aim is to enable a robot to interact with its environment through a set of Vision Algorithms. The ultimate product is a fully functional Robotics Vision system where various mechanical parts can be placed in arbitrary positions in the field of view of the robot and the robot can grasp or move them either according to a set of predetermined rules or with user intervention.

The use of Vision Applications has become increasingly attractive in factory automation settings. This is due to several factors such as:

- Ability to manufacture products faster than manual means.
- Ability to repeat the manufacturing procedure consistently over time.
- Ability to reduce or eliminate recurring labor costs via easy justification of investment.
- Ability to use the same conveyors, trays, bins, totes, racks, etc. when products change.

Therefore it is apparent that automating various tasks in factory settings has multifold benefits including cost saving and higher efficiency. However developing a fully operational Vision System for a real-world application, which is the aim of this thesis, requires an algorithm with a high degree of flexibility and fault tolerance.

Another important factor in such a system would be the time constraint. This is one constraint that can not be ignored by any vision algorithm if real-world applications are desired. This is to say that a certain task has to be accomplished by the algorithm within a given amount of time or undesirable consequences will occur. This is why we have explored the Hopfield Neural Network algorithm (HNN). As will be explained later on, Neural Networks (NN) are parallel processing algorithms that take advantage of many processing units that work simultaneously. This implies that hardware implementation of algorithms that depend on NN can produce extremely efficient mechanisms in terms of time.

1.2 Motivation

As explained in the previous section there are many reasons why an assembly line task should be automated using a vision system. However, the important factor to consider is that vision systems are often not as accurate as their human counterparts. That is to say that the error rate of a vision system can be prohibitively larger than what is expected in a certain application. Another important factor in the design of vision systems is the time constraint as mentioned above. Although much research has been conducted in the field of vision, still most algorithms are too computationally expensive to be realistically implemented in actual factory settings.

The aim of this thesis is to develop a faster and more robust vision system for a bin-picking application. This has been achieved by introducing the use of HNN for the task of stereo matching which will be thoroughly explained in the next few sections. This design has the potential of real-time hardware implementation. It is also worth noting that HNNs have never been implemented in Machine Vision applications (to the author's knowledge). Therefore this work is an experiment in the use of an unpopular, yet potentially useful tool for achieving real-time stereo matching. In fact there have been few works [1-3] where the feasibility of HNN as stereo matching tools in vision applications have been considered. As mentioned, none have explored this in a machine vision application. Moreover, HNNs were in fact devised originally as associative

memory [4]. However it will be shown that HNNs can potentially be very advantageous for certain combinatorial optimization problems. This and other aspects of HNN will be discussed in chapter four.

1.3 Thesis Organization

Following the introduction, the second chapter offers preliminary information that is required for the camera calibration chapter. The second chapter offers an introduction to projective geometry which is the tool that is used to describe many computer vision problems. It is necessary to present various concepts regarding projective geometry in order to discuss the problem of camera calibration. Amongst other things, this chapter explains the projection matrix and the 3D to 2D projection which occurs during any image formation. Also this chapter explains various concepts regarding multiple camera geometry such as the fundamental matrix. Again this is critical for understanding the epipolar geometry which will be thoroughly discussed in chapter five.

Following this, the third chapter discusses the actual camera calibration process used in this thesis which utilizes the concepts presented in chapter two. Various error tolerance factors and also the robustness of the algorithm will be discussed and results will be presented. It is important to note that the calibration step is as critical as the 3D reconstruction step, since without accurate calibration results it is impossible to acquire accurate 3D measurements. Thus a significant amount of space is devoted to discussing the calibration step and how more accurate results can be obtained.

The fourth chapter discusses the HNN. This is a brief introduction to HNNs and also how they can be used as a stereo matching tool. Their operation and advantages and disadvantages will be discussed. Also some comparisons will be made on the use of HNN as opposed to the traditional method of stereo matching.

Chapter five introduces the stereo matching process. This chapter discusses the fundamentals and also the stereo matching constrains, which are an important topic is devising a stereo matching algorithm. Following this, some traditional methods for stereo matching will be presented. In addition, the state of the art in stereo matching will be

discussed in this chapter. And finally chapter six details the use of HNNs in stereo matching, which is the main contribution of this thesis.

Obviously, after 3D information is acquired this data has to be used in some sort of a control application where an articulating arm or other mechanical manipulator will interact with some object(s) in the environment. Therefore it is important to discuss how the raw 3D data obtained from stereo matching can be used to accomplish the actual end goal of the vision system. This goal could be anything ranging from moving a conveyor belt to picking objects to sending various commands to an autonomously navigated vehicle. In this thesis the final result to be achieved is grabbing of a number of mechanical objects from the field of view. Although these objects are marked, the algorithm and programs are extendible to any arbitrary environment.

Finally chapter seven presents the conclusions of this work. It is important to note this thesis is not merely a work in stereo matching (although that is the focus). Rather a Vision System has been presented where several stages have been incorporated into the work as shown in Figure 1.1. It is often the case where stereo vision research is concentrated on dense matching algorithms where researchers conduct their work on stereo image databases [5] and devise their algorithm without consideration for actual vision systems where those stereo algorithms can potentially be implemented. However this work includes research and code in calibration, 3D data analysis and robotics control and stereo matching, all integrated into one working Vision System. Finally the appendix contains the C++ source code to some of the core classes used to simulate the algorithm.

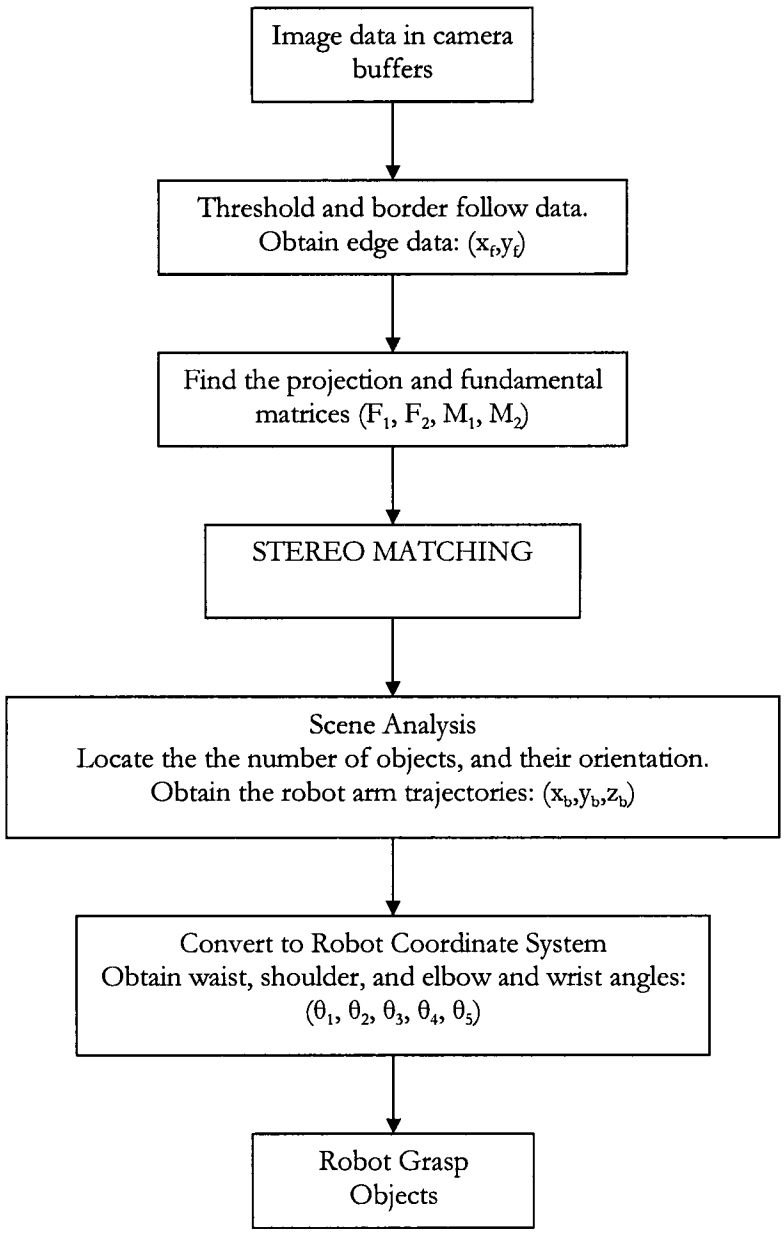

Figure 1.1 Various steps required for the transformation of image data into the appropriate combination of robot joint angles for grasping objects.

Chapter 2: 3D Vision

2.1 Introduction

3D vision is the science of acquiring 3D information regarding objects within some field of view. Although this thesis concentrates on 3D vision using stereo, there are various other methods used for acquiring 3D information [6]. The next sections will be discussing the mathematics and geometry behind stereo vision, however here a brief introduction to other 3D vision methods will be provided. Also the justification behind choosing stereo over other methods will be presented.

2.1.1 Active versus Passive Vision

Several approaches have been developed for 3D vision [6]. They can be broadly categorized into two major classifications: active vision systems and passive vision systems. In active vision systems (Figure 2.1) structured light (e.g., laser) highlights the points on the object to be measured.

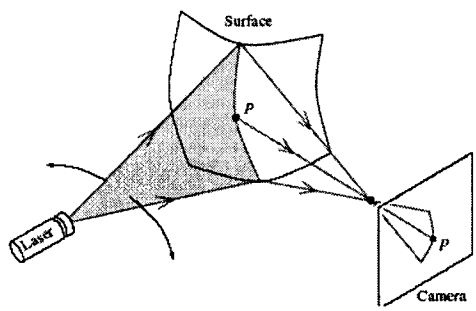

Figure 2.1 Active vision using triangulation

Note that there are many other methods for determining 3D properties of objects in images such as [6]:

- Shape from shading

- Shape from motion
- Shape from optical flow
- Shape from texture
- Shape from focus/defocus

2.2 Projective Geometry

It is vital to give a brief introduction to various elementary topics in computer vision before the rest of the thesis can be presented. First and foremost, projective geometry is an essential tool in order to understand higher level vision concepts. Note that much of the information here has been obtained from [7] and [8].

2.2.1 Euclidean versus Projective Geometry

We are all familiar with Euclidean geometry and with the fact that it describes our three-dimensional world so well. In Euclidean geometry, the sides of objects have lengths, intersecting lines determine angles between them, and two lines are said to be parallel if they lie in the same plane and never meet. Moreover, these properties do not change when the Euclidean transformations (translation and rotation) are applied. Since Euclidean geometry describes our world so well, it is at first tempting to think that it is the only type of geometry. (Indeed, the word geometry means ``measurement of the earth''). However, when we consider the imaging process of a camera, it becomes clear that Euclidean geometry is insufficient: lengths and angles are no longer preserved, and parallel lines may intersect.

Euclidean geometry is actually a subset of what is known as projective geometry. In fact, there are two geometries between them: similarity and affine. Projective geometry models well the imaging process of a camera because it allows a much larger class of transformations than just translations and rotations, a class which includes perspective projections. Of course, the drawback is that fewer measures are preserved -- certainly not lengths, angles, or parallelism. Projective transformations preserve type (that is, points

remain points and lines remain lines), incidence (that is, whether a point lies on a line), and a measure known as the cross ratio, which will be described.

2.2.2 Imaging as a Projective Transformation

Projective geometry exists in any number of dimensions, just like Euclidean geometry. For example the projective line, which we denote by P^1, is analogous to a one-dimensional Euclidean world; the projective plane, P^2, corresponds to the Euclidean plane; and projective space, P^3, is related to three-dimensional Euclidean space. The imaging process is a projection from P^3 to P^2, from three-dimensional space to the two-dimensional image plane. And this is where projective geometry help explains this process and furthermore help model this transformation (which is used in camera calibration, and 3D reconstruction).

2.3 Homogeneous Coordinates

Another important concept in the field of computer vision is homogenous coordinates. This is a rather different method of representing points and lines in space as opposed to the Euclidean coordinate system which most people are used to.

Suppose we have a point (x,y) in the Euclidean plane. To represent this same point in the projective plane, we simply add a third coordinate of 1 at the end: (x, y, 1). Overall scaling is unimportant, so the point (x,y,1) is the same as the point (αx, αy, α), for any nonzero α.

$$(X,Y,Z) = (\alpha X, \alpha Y, \alpha Z) \tag{2.1}$$

In other words, for any $\alpha \neq 0$ (thus the point (0,0,0) is disallowed). Because scaling is unimportant, the coordinates (X,Y,W) are called the homogeneous coordinates of the point. For example, to represent a line in the projective plane, we begin with a standard Euclidean formula for a line:

$$ax + by + c = 0 \tag{2.2}$$

and use the fact that the equation is unaffected by scaling to arrive at the following:

$$\begin{aligned} aX + bY + cW &= 0 \\ u^T p &= p^T u = 0 \end{aligned} \tag{2.3}$$

where $u=[a,b,c]^T$ is the line, and $p=[X,Y,W]^T$ is a point on that line. Thus we see that points and lines have the same representation in the projective plane. The parameters of a line are easily interpreted: -a/b is the slope, -c/a is the x-intercept, and -c/b is the y-intercept.

2.3.1 Conversion from Projective to Euclidean Coordinates

To transform a point in the projective plane back into Euclidean coordinates, we simply divide by the third coordinate: (x,y) = (X/W, Y/W). Immediately we see that the projective plane contains more points than the Euclidean plane, that is, points whose third coordinate is zero. These points are called ideal points, or points at infinity. There is a separate ideal point associated with each direction in the plane; for example, the points (1,0,0) and (0,1,0) are associated with the horizontal and vertical directions, respectively. Ideal points are considered just like any other point in P^2 and are given no special treatment. All the ideal points lie on a line, called the ideal line, or the line at infinity, which, once again, is treated just the same as any other line. The ideal line is represented as (0,0,1).

2.4 Camera Model

Using the information provided in the section on projective geometry and homogenous coordinates, we are now ready to present the basic camera model which is an essential concept in stereo correspondence and camera calibration. It is important to explain the camera model before any further information about calibration or 3D reconstruction can be provided.

2.4.1 Projective Camera

There are a few different models for describing a camera, however the one mostly used in this thesis is the projective camera (others include affine, pinhole, camera at infinity ...). A general projective camera can be represented by a matrix, *P*. The camera will map world points, X to image points x according to x=PX. This matrix P, is also referred to as the projection matrix which will be thoroughly discussed in chapter three. In homogeneous coordinates we can write a projective camera as ΠA, where A is a general projective transformation and Π is a perspective camera transformation. For reference, this means that

$$\Pi = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{2.4}$$

and

$$A = \begin{pmatrix} a_{00} & a_{00} & a_{00} & a_{00} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \tag{2.5}$$

Notice that, because we are working in homogenous coordinates, A and αA represent the same transformation if $\alpha \neq 0$. Note that the matrix A can be further broken down, this will be discussed later in chapter three.

2.4.2 Lens Distortion

There are various nonlinearities in these equations that are ignored since their inclusion will incur additional computational complexity and yet would not add significant accuracy increase. In brief, lens distortion is defined as failure of the lens to image a straight line in object space as a straight line in image space and to maintain the same metric. Normally, the resultant displacement of the image from that produced by a

true projective transformation is referred to as lens distortion and is characterized by two components. One that is radically symmetric about the principle point and one that is symmetric along a line directed through the principle point [9]. Terminology adopted by photogrammetrists to describe these two components is radial and tangential distortion, respectively. Since distortion has to affect the position of the image point in the image plane, therefore represents a factor in camera calibration parameters, and should be compensated for by using various correction schemes. For a complete analysis of lens distortion and its affects on camera calibration see the work by R. Tsai [10].

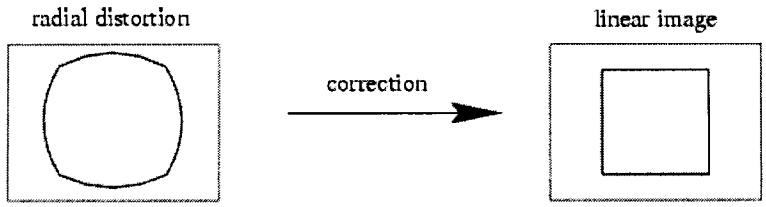

Figure 2.2 Effects of lens distortion on images and their correction.

As explained, these nonlinearities are results of imperfect lens characteristics. Note that another reason for the exclusion of nonlinear lens behavior due to distortion is the fact that improved manufacturing techniques in lens technology has lead to a decrease in this phenomenon. This is to say those newer lenses are not as prone to the effects of lens distortion as the ones manufactured before.

Chapter 3: Camera Calibration

3.1 Introduction

As explained in the previous section, the camera can be represented in a matrix format. This format can in fact be further decomposed into its constituent parts. The goal of camera calibration is to find this matrix. The reason for this is that once stereo matching takes place, it is impossible to find the 3D information regarding the objects without the aid of this matrix. In fact, this matrix contains information on how the pixels in an image are related to some outside 3D coordinate system. It is using this matrix that the 3D information can be recovered.

Another use for the calibration step is to find the epipolar geometry which will be explained later in chapter five. This helps the stereo matching algorithm narrow its search neighborhood significantly and thus helps in improving the performance of the system. Note that calibration is the process of finding the projection matrix rather than the epipolar geometry (represented by the fundamental matrix); however, the epipolar geometry can also be found at this stage. This makes the calibration step even more important. Also calibration has to be performed only once and the data can be used in subsequent image snapshots. However, once the cameras have moved (even a slight movement) they have to be recalibrated in order to have accurate information regarding the 3D data.

In order to better understand the reason behind camera calibration lets take a closer look at the camera projection formula:

$$\begin{pmatrix} uw \\ vw \\ w \end{pmatrix} = \begin{pmatrix} a_1 & a_2 & a_3 & a_4 \\ a_5 & a_6 & a_7 & a_8 \\ a_9 & a_{10} & a_{11} & a_{12} \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \tag{3.1}$$

This equation describes how a 3D point maps to a 2D point (u, v) in the image plane. Therefore when looking at pixels in a camera, the information that is known about them is their image coordinates which is (u, v) multiplied by a scale factor w. It is obvious that if we were given the projection matrix (a1, a2...) and the 3D world coordinates of a point we could easily find its image points using a simple matrix multiplication. However, the opposite is not true, that is, since one axis is lost during this transformation, if given an image point, and we can not uniquely find its corresponding 3D point. This is where stereo vision gives us a second set of equations from a second image which leads to an over determined set of equations that can easily be solved for the world coordinates (X, Y, Z). However throughout this process it is obvious that without the projection matrix our goal of finding the 3D world coordinates will not be achieved.

3.2 Projection Matrix

It is important to better understand the projection matrix before proceeding to explain the camera calibration. We explained before that the projection matrix can be broken to two parts. However, there are really three parts to the projection matrix.

3.2.1 Decomposition of the Projection Matrix

The projection matrix is composed of the following parts:

1- A 3D Euclidean transformation: This is a 3D rigid displacement where a scene points, initially defined in a scene reference frame, is transformed so that they would be defined in the camera reference frame. This transformation has 6 parameters corresponding to a 3D rotation and a 3D translation.

2- A 3D-2D transformation: 3D points defined in the camera reference frame are projected on the image plane. The new coordinates of these new points are called normalized coordinates.

3- A 2D-2D transformation: The normalized coordinates expressed in the scene metrics, undergo a 2D affine transformation to become defined in pixels in the image plane reference frame.

This three step transformation is best demonstrated by Figure 3.1.

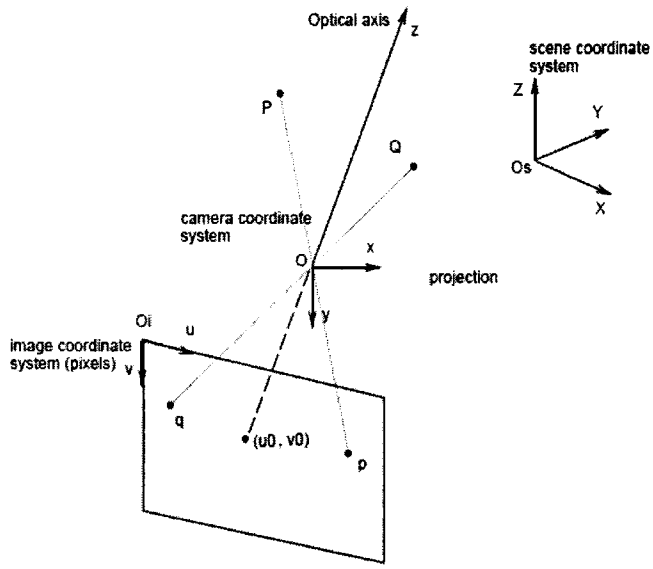

Figure 3.1 The three step coordinate system transformation of the perspective camera

These three transformations can be algebraically explained. First, the 3D Euclidean transformation. Given a point P in the world coordinate system, and its location P' in the camera reference frame, where P=(X, Y, Z) and P'= (X', Y', Z'), their relationship can be described as:

$$\begin{pmatrix} X' \\ Y' \\ Z' \end{pmatrix} = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} \tag{3.2}$$

Using homogenous coordinates we can write the above in a compact way as follows:

$$\begin{pmatrix} X' \\ Y' \\ Z' \\ 1 \end{pmatrix} = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \tag{3.3}$$

or simply P'=DP.

Moving on, the 3D-2D transformation is the next step of this transformation. Keep in mind that the point undergoing these transformations is a fix point and these transformations are only operating on the coordinate system that this point is defined in. This second transformation (3D-2D) is the projection that a point in the 3D space undergoes to become a point in the 2D space on the image plane. This is where one of the axes is lost and needs to be recovered later on to achieve 3D reconstruction. The following figure demonstrates this phenomenon.

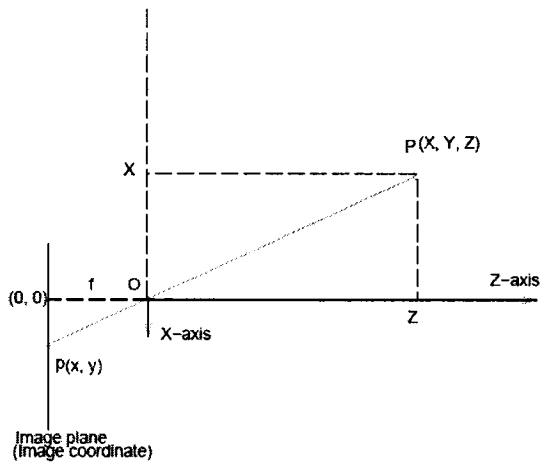

Figure 3.2 Perspective projection of a point.

The new coordinate values can in fact be found using similar triangles:

$$\frac{Z}{f}=\frac{X}{x}\rightarrow x=f\frac{X}{Z} \quad and \quad \frac{Z}{f}=\frac{Y}{y}\rightarrow y=f\frac{Y}{Z} \tag{3.4}$$

where f is the focal length. This relationship is in fact a perspective projective transformation from P^3 to P^2. The following equation demonstrates this assuming the focal length, f is unity:

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \lambda \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \tag{3.5}$$

A compact version of the above will be: $p=\lambda IP$, where I is a 3x4 matrix representing a perspective projection and, λ is a scale factor for equality in the projective space. The perspective projection of a point P, defined in a scene coordinate system, on a plane at distance 1 from the optical center is given by : $p=\lambda IDP$, where, D takes care of changing the coordinates from the scene reference frame to the camera (projection system) reference frame and I takes care of the perspective projection.

Following this the 2D-2D transformation will take place. This is due to the fact that after projection, points are defined in the scene units, for instance centimeters. However, the unit, used for image points is the pixel, therefore a scaling is necessary. To illustrate this point, consider a (x, y, 1) be the coordinates of p, a point located on the image plane, expressed in the scene unit. And let (u, v, 1) be the coordinates of p, a point located on the image plane, expressed in pixels. The issue becomes of determining the relationship between the two. Moreover, the scaling is not the only problem that has to be accounted for. Another issue is the fact that the current origin is located at (u_0, v_0), the intersection of the optical axis with the image plane. But the origin of the pixel coordinates should be the upper left corner of the image. Therefore the scaling has to be following by an addition. The following demonstrates this relationship which contains both the scaling and the addition:

$$A = \begin{pmatrix} \alpha_u & 0 & u_0 \\ 0 & \alpha_v & v_0 \\ 0 & 0 & 1 \end{pmatrix}$$
$$\begin{pmatrix} u & v & 1 \end{pmatrix}^T = A \begin{pmatrix} x & y & 1 \end{pmatrix}^T \tag{3.6}$$

where A is the affine transformation of the 2D space, and α_v and α_u are defined as:

$$\begin{aligned} \alpha_u &= -fk_u \\ \alpha_v &= -fk_v \end{aligned} \tag{3.7}$$

where f is the focal length in mm and k_u and k_v are the number of pixels per mm along the Y and X axis.

After combining the above results we can arrive at the precise definition of the projection matrix:

$$M = AID$$
$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \lambda \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \tag{3.8}$$

where (u, v) is the image point, (X,Y,Z) is the world point and λ is a scale factor.

3.2.2 Intrinsic and Extrinsic Parameters

From the above it can be seen that the projection matrix encodes information regarding two different set of parameters. One set is that of the camera, such as focal length and the number of pixels per millimeter and the camera centre (u_0, v_0). These are called the intrinsic parameters of the camera and are independent of the position of the camera related to the world coordinate system. The other set of parameters are those described by matrix D, which was explained in equation(3.3). These parameters are directly related to the position of the camera. Although these parameters do not have to

be necessarily found explicitly (finding projection matrix, M is sufficient for our application) but some simplifications can be made if these parameters need to be found explicitly. For instance, intrinsic parameters can be found and later on only the extrinsic parameters need to be estimated since generally the intrinsic parameters do not need to be estimated more than once. Another simplification methods is to eliminate the extrinsic parameters altogether by fixing the world coordinate system at the camera reference frame (so positions are estimated relative to the actual camera). This would be very effective in cases where cameras need to move. However, in this project only the matrix M is found since explicitly finding the extrinsic/intrinsic parameters would not be useful since we need a coordinate system which is independent of the camera. This is because a robot needs to get the information from the camera and interact with the environment. Thus fixing the coordinate system at the camera would not be useful in this case. However for applications such as autonomous navigation fixing the world coordinate system to move with the camera is the only possible solution.

3.2.3 Estimation of the Projection Matrix

By finding a number of image points whose 3D world coordinate locations are known we can rearrange the equation(3.8). This can be simply written as an over determined set of linear equations if the number of such points is higher than six and can be solved using any number of numerical methods such as Singular Value Decomposition (SVD) [11].

$$\begin{pmatrix} -X_1 & -Y_1 & -Z_1 & -1 & 0 & 0 & 0 & 0 & u_1X_1 & u_1Y_1 & u_1Z_1 & u_1 \\ 0 & 0 & 0 & 0 & -X_1 & -Y_1 & -Z_1 & -1 & v_1X_1 & v_1Y_1 & v_1Z_1 & v_1 \\ -X_2 & -Y_2 & -Z_2 & -1 & 0 & 0 & 0 & 0 & u_2X_2 & u_2Y & u_2Z_2 & u_2 \\ 0 & 0 & 0 & 0 & -X_2 & -Y_2 & -Z_2 & -1 & v_2X_2 & v_2Y & v_2Z_2 & v_2 \\ & & & & & \cdot & & & & & & \\ & & & & & \cdot & & & & & & \\ & & & & & \cdot & & & & & & \\ -X_n & -Y_n & -Z_n & -1 & 0 & 0 & 0 & 0 & u_nX_n & u_nY & u_nZ_n & u_n \\ 0 & 0 & 0 & 0 & -X_n & -Y_n & -Z_n & -1 & v_nX_n & v_nY & v_nZ_n & v_n \end{pmatrix} \begin{pmatrix} m_{11} \\ m_{12} \\ m_{13} \\ m_{14} \\ m_{21} \\ m_{22} \\ m_{23} \\ m_{24} \\ m_{31} \\ m_{32} \\ m_{33} \\ m_{34} \end{pmatrix} = 0 \quad (3.9)$$

In order to actually find the matrix shown above and to estimate the projection matrix of each camera, M_1 and M_2, the calibration process must be undertaken. This is done by finding several image locations and their 3D coordinates as mentioned above using these summarized steps:

- Place a calibration target in the view of the cameras
- Take snapshots from each camera
- Change location of the calibration target to several different heights, take snapshots, repeat as many times as necessary
- Detect feature points of the calibration target (the locations of these feature points must be known in the world coordinate system that is to be used in the calibration)
- Use a numerical method to find the projection matrix for each camera

Two terms need defining at this point, the first one being the "calibration target". This is a physical object which will be placed in some known coordinates, where the user would note the location of each of its feature points. To clarify, let's look at the calibration target used in this thesis in Figure 3.3. This object is moved to various heights, and the centroids of the circles are used as features.

It is important to know that we must move the calibration target to at least two different heights, because using feature point on a plane only would lead to a degenerative configuration [8] and the numerical solution to the projective matrix would be incorrect. In this thesis we decided to move the object to six different heights, this does lead to a high degree of redundancy and thus a higher quality solution is obtained.

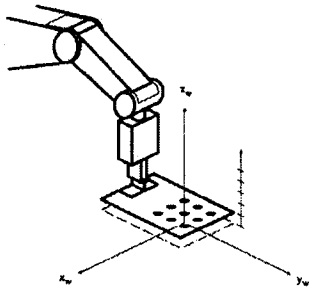

Figure 3.3 Calibration target

However, the calibration in this thesis is somewhat inaccurate compared to other works, mainly because a robot is used to move the calibration target, and since the movements of the robot are error prone, a highly accurate projection matrix could not be obtained.

3.2.4 Calibration Targets

Note that it is possible to use any kind of calibration target. For instance a more popular calibration object than the one used in his thesis is the one using square objects such as the one shown in Figure 3.4. The reason why this target is more popular is that every object in this case (squares) provides four feature points which are the corners instead of one feature per shape as is the case with circular objects.

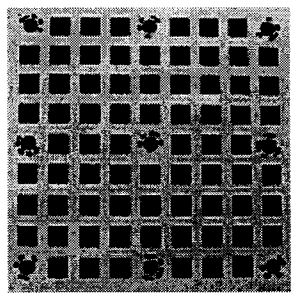

Figure 3.4 Other forms of calibration targets

These corners can in fact be very accurately extracted using various methods including the corner detector [12].

A more interesting calibration method is to use two perpendicular planes. This would mean that the calibration target does not have to move. This is highly desirable because the movement of the calibration target is one of the sources of error in the calibration. So only a single snapshot of such a calibration target as shown in Figure 3.5 could lead to a very efficient calibration scheme, note the number of squares, each provides four points that can be used in the calibration process. Now let's move on to the actual calibration process using the target of Figure 3.3. After the robot moves the calibration target to a new known location, a snapshot is taken. Afterwards several steps have to be taken. First, using a simple thresholding method, the image is thresholded. In this case the Otsu threshold was used [13]. This algorithm is based on a very simple idea which is to find the threshold that minimizes the weighted within-class variance. It is fast and simple and suitable to this situation. Following this a border following algorithm is performed on the binarized image. The purpose of this step is to effectively isolate the circles on the calibration target in order to pin point their exact location in the images.

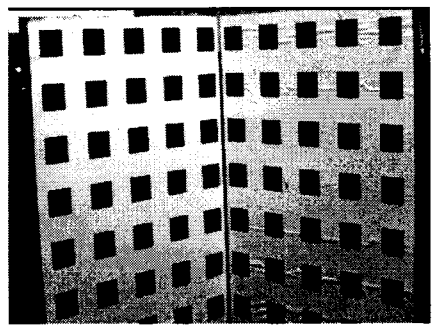

Figure 3.5 A more effective calibration target

One important point is that in this project the border following was severely hindered by the presence of spurious reflection off of the robot surface which is a shiny metallic surface. In order to avoid this, a simple shape descriptor was used (normalized distance of border points to the centroid). And this feature was checked for every object

in order to ensure the shapes that are extracted are perfectly circular. Note that the size is not a good measure in this case since the calibration target moves to several different heights in respect to the camera and thus its projection on images changes as a result of this different in the distance of the viewing camera.

3.2.5 Implementation of the Calibration Stage

To better see how this works in reality several snapshots of the program used is shown. The first step is taking the images and binarizing them as shown in Figure 3.6. This screen is shown to the user for every time the calibration target moves to a new height. Therefore it is shown six times. The first row images are those of the snapshot from the calibration target. The second row shows the thresholded version of the calibration target using the Otsu method. Here you can see that the thresholding picks up objects that do not belong to the calibration target; these are parts of the robot which were referred to earlier as spurious reflections.

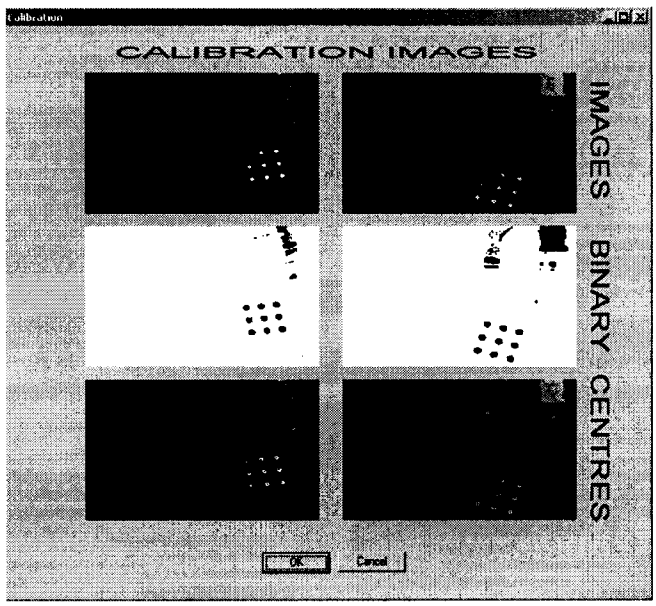

Figure 3.6 Calibration software – Thresholding and border following followed by centroid calculation.

The third row shows the actual images, plus the centroid of the calibration circles (these marks are one pixel wide). It is possible for the user to intervene if the centroids have not been picked up correctly. In brief, the screenshots of the software shows five steps, the grabbing of the frame (snapshot), the thresholding, border following, shape detection and centroid detection.

The second screenshot shows the process of establishing correspondence between the detected circles and those of the actual calibration target. This involves user intervention although it is possible to perform this automatically such as done by Zhang [14]. Since the equation(3.9) needs the actual location of the calibration features plus their location in the image, these two entities need to be paired together. As previously mentioned, the world coordinate locations of the features are known, and the image location of the features (centroids) is also detected using the procedure outlined earlier. All that is left to do is to pair these two values together so that they can be used in equation(3.9). This is done using software and interacting with the user by guiding him/her as shown in Figure 3.7.

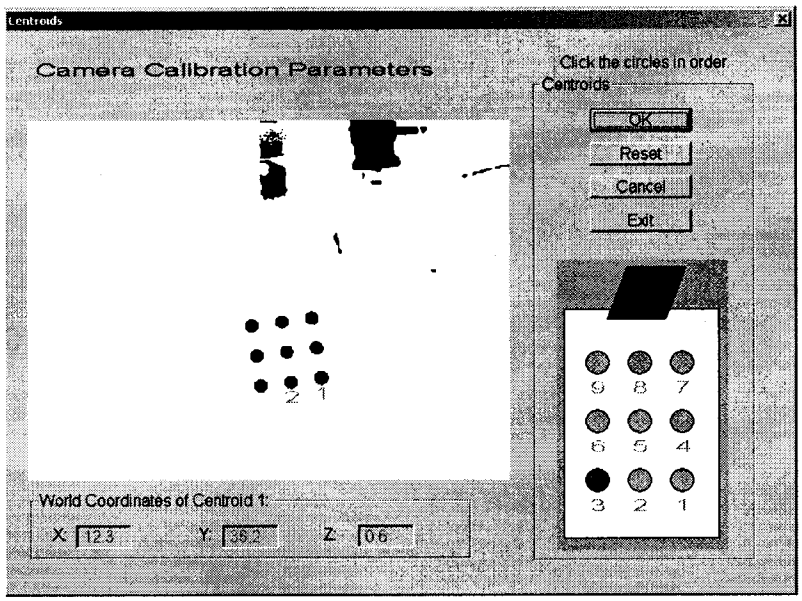

Figure 3.7 Pairing the image locations with world locations by user intervention.

A computer generated image of the calibration target is used where the user is shown one of the circles blinking. The software starts out by blinking the first circle (first circle could be any). Afterwards the user clicks the circle that corresponds to this blinking circle. At this stage the circles are detected and the software is aware of them therefore the user need only click in the vicinity of the first circle. After the user clicks the first circle the software starts blinking the second circle and so on. At every turn the user clicks the corresponding circle. This is obviously tedious and inconvenient since the calibration target has nine circles, and it has to move to size heights, and at every height two images are taken. This means the user has to click 108 circles. This it not prohibitively inconvenient since calibration is a one time process (one time until the cameras move). However alternative calibration methods are recommended to the reader to avoid user intervention. For instance the calibration technique using the target of Figure 3.5 can be done using minimal user intervention and by using homographies [8].

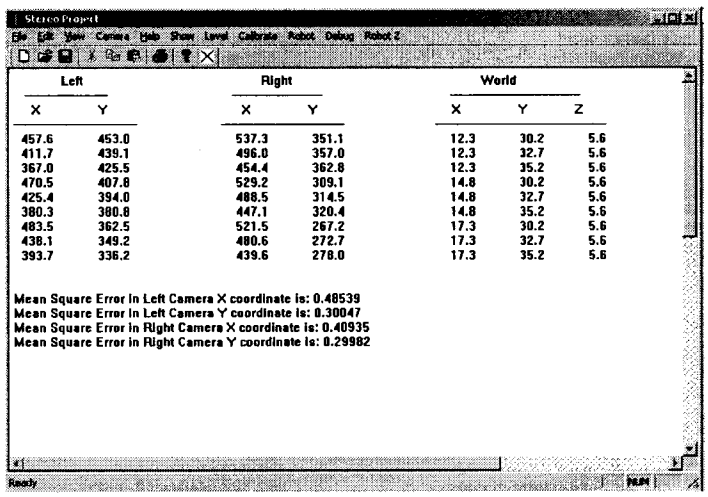

Figure 3.8 Calibration results shown to the user.

Once the user finishes selecting the circles in order, the program forms the matrix of equation (3.9) and solves the system using SVD. Following this the results of the calibration are printed out on the screen. Also several text files are printed out that

contain the calibration data. As explained the calibration and stereo matching and robot grabbing modules are separate entities. The calibration module uses text data files in rode rot relay the calibration to other modules in the application.

The next screenshot is the one in Figure 3.8 which can be divided into two parts. The top part shows the actual image location of the feature points in both cameras (left and right in the top part denote left and right cameras). That is why there are nine rows in the top part. On the top right part we can see the world locations of these nine feature points. There could be six different such screen for the top part for every single movement of the calibration target. Finally the bottom part contains information regarding the accuracy of the projection matrix in pixels for each camera. This information is obtained by using the projection matrix which was obtained from SVD to recalculate the image locations of the feature points from the 3D (world coordinate) points of their coordinate in space. Although we have the image locations of the features (found using thresholding, border following and ...) we can recalculate them numerically using the estimated projection matrix and then compare the two together. The closer they are indicates the more accurate the projection matrix is. This is how the information provided in Figure 3.8 shows a measure of the accuracy of the projection matrix. Note that here the error is presented in terms of pixels, a better method of assessing the accuracy of the projection matrix is to actually recalculate the world coordinates using the projection matrix, this way the error measures will be in terms of more useful metric such as millimeters instead of pixels (which could translate to any measure in the real work depending on the camera and scene geometry). It can be seen that the error is less than half a pixel approximately in the X and Y (image axis) coordinates, which is a reasonable accuracy to work with.

3.2.6 Accuracy of the Calibration Data

Keep in mind that without accurate camera calibration, it is impossible to achieve high accuracy in stereo matching and in 3D reconstruction using the stereo matches. This is because a poorly calibrated camera will produce an inaccurate projection matrix which leads to incorrect 3D coordinate calculations, and it will create an incorrect fundamental

matrix which means it is going to produce an incorrect epipolar line and so this essential constraint can not be used in stereo matching (more on this in chapter five). This means that we need other ways of ensuring consistency and accuracy of the calibration results. One way is to take a closer look at the reprojections of the image features and their actual locations such as in Table 3-1. This helps in debugging purposes if for instance; one of the axes is creating a higher error rate. Another way to ensure consistency of the calibration data is to calculate the location of the camera centers. The location of the camera centers is found using a simple equation: MC=0.

Table 3-1. Calibration data and reprojections

Error in X	Error in Y	U	V	Level	Point #
0.378	0.259	264.717	437.302	1	1
0.053	0.062	214.216	432.863	1	2
0.059	0.147	163.981	428.115	1	3
0.003	0.145	268.08	386.52	1	4
0.000	0.118	218.176	381.784	1	5
0.230	0.2	167.765	377.314	1	6
0.294	0.294	272.098	335.784	1	7
0.335	0.093	221.725	331.098	1	8
0.162	0.074	172.204	326.204	1	9
0.076	0.269	282.065	419.419	2	1
0.248	0.231	227.746	414.322	2	2
0.111	0.144	173.565	409.145	2	3
0.163	0.475	285.807	364.772	2	4
0.048	0.221	232.213	359.311	2	5
0.136	0.565	177.903	354.419	2	6
0.096	0.35	289.831	310.153	2	7
0.343	0.148	235.967	304.6	2	8
0.270	0.15	182.644	299.22	2	9

3.2.7 Finding the Camera Centre

Another way to state this formula is to say that the camera centre is the right null-space of M (the projection matrix). It might appear at first glance that this equation states that the centre of the camera is a location in the 3D world coordinate system that is

projected on the location zero $[0, 0]^T$ in the image (the upper left corner) which might seem somewhat arbitrary. However this is not true, because all equations up to this point have been defined in homogenous coordinates, therefore point zero is actually point $(0, 0, 0)^T$ in the two dimensional image space, which is in fact an undefined point in the projective space. This is intuitive because it would make perfect sense if the centre of the camera was projected onto an undefined location.

The reason for finding the camera centre is mostly to ensure the calibration is consistent with our pre-determined world coordinate system. This is necessary since it is possible through some kind of error to obtain calibration matrices with very low error rate, but inconsistent with our world coordinate system. This is possible for instance if there is a systematic error in measurement error when providing the 3D coordinates of feature points of the calibration matrix to the software. For instance if every x coordinate system has the same offset, the projection matrix will be accurate but it will provide a calibration matrix for a biased coordinate system. Therefore by finding the camera centers we can check the locations of the camera to the world coordinate system (by hand) and ensure these results reflect our desired world coordinate system. In other words finding the camera centers is a simple sanity check for ensuring the consistency of the coordinate system. For instance a typical camera centre location which is outputted as a flat text file in our simulations has the form:

C_1=[8.64476 ; 47.96822 ; 71.98662 ; 1.00000]

C_2=[8.10730 ; 29.15020 ; 71.19551 ; 1.00000].

Note the extra one at the end, again this is described in homogenous coordinates. In our experiments we measure the location of the camera with respect to the world coordinate system and we obtained similar results as those provided by the software.

3.3 The Fundamental Matrix

One of the most important things to consider when performing stereo matching is the epipolar geometry. This will be explained later on, however its calculation is done in

the calibration stage and thus its algebraic representation (the fundamental matrix) is discussed here.

Let two images be taken by two cameras by linear projection, as shown on Figure 3.9. Let O and O' be the two projection centers of the two images which will be called in the following: first and second image respectively. The point O projects to the point e' in the second image, and the point O' projects to the point e in the first image. The two points e and e' are the epipoles and the lines through e and e' are the epipolar lines. Let a space point P be projected on p and p' respectively in the first and the second image. The plane defined by the three points P, O and O' is the epipolar plane, it contains the two points p and p'. The projections of this plane onto the first and the second image are respectively the epipolar lines (e, p) and (e', p'). We can see from Figure 3.9 that the relation between the epipolar lines in the first image and the epipolar lines in the second image is a homography of lines. This homography is the epipolar transformation which relates a pair of stereo images. Note that a homography is a geometric transformation relating two planes, for more information the reader can refer to a number of references including Hartley and Zisserman [8].

The most common way to describe the epipolar geometry is by means of a 3 x 3 matrix called the fundamental matrix. This matrix, usually denoted F, contains the geometric information which relates a pair of images. For a point p given by its homogeneous coordinates in the first image, the corresponding epipolar line l_p, in the second image is given by:

$$l_p = Fp \tag{3.10}$$

where l_p is a vector containing the coefficients of the line. If in the second image p' is the corresponding point to p then p' must belong to l_p this can be written as:

$$p'^T Fp = 0 \tag{3.11}$$

F is homogeneous and is of rank two [8], it has seven independent parameters.

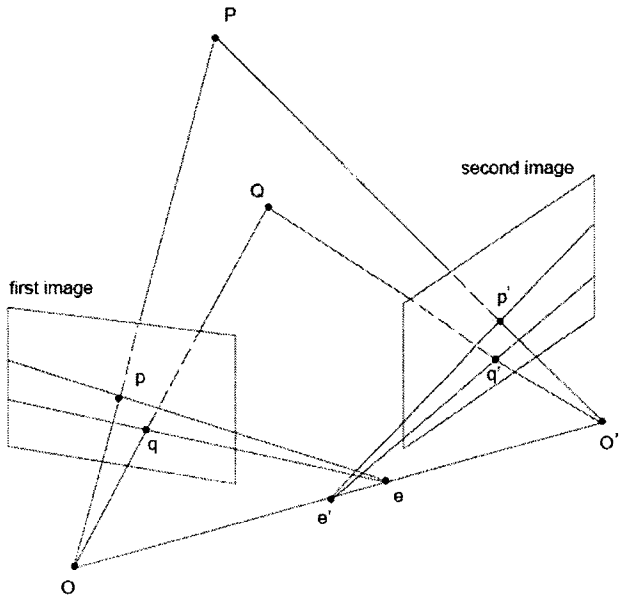

Figure 3.9 A space point P defines the epipolar plane OPO' which intersects the two image planes in two lines (ep) and (e'p'): the epipolar lines.

3.3.1 Estimation of the Fundamental Matrix

The most popular methods to compute F are based on equation(3.11). A brief description of these methods follows. Let's assume that points have been matched in the two images. When using equation(3.11), each couple of match points (p,p') gives rise to a linear homogeneous equation in the nine unknowns of F. Since F can only be defined up to a scale factor, it has 8 independent parameters. So F can be computed with 8 matched points in the two images. When more than 8 matches are given, a linear least square method can be used to compute F. In this thesis, the SVD is used for solving an over determined set of equations [8]. This numerical method is particularly attractive for its including noise cancellation abilities. The equation to be solved by SVD can be shown as follows:

$$\begin{pmatrix} u_1'u_1 & u_1'v_1 & u_1' & v_1'u_1 & v_1'v_1 & v_1' & u_1 & v_1 & 1 \\ u_2'u_2 & u_2'v_2 & u_2' & v_2'u_2 & v_2'v_2 & v_2' & u_2 & v_2 & 1 \\ & & & & \cdot & & & & \\ & & & & \cdot & & & & \\ & & & & \cdot & & & & \\ u_n'u_n & u_n'v_n & u_n' & v_n'u_n & v_n'v_n & v_n' & u_n & v_n & 1 \end{pmatrix} \begin{pmatrix} f_{11} \\ f_{12} \\ f_{13} \\ f_{21} \\ f_{22} \\ f_{23} \\ f_{31} \\ f_{32} \\ f_{33} \end{pmatrix} = 0 \qquad (3.12)$$

This is done by obtaining n number of correspondences between the two images which is done during the calibration stage. To see this more clearly lets revisit equation(3.11). This relationship can be better written as:

$$\begin{pmatrix} u' & v' & 1 \end{pmatrix} \begin{pmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{pmatrix} \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = 0 \qquad (3.13)$$

Where the f values represent members of the fundamental matrix. Therefore by combining a set of known relationships in corresponding points in the two images we can construct a simple system that can be written in the form of

$$AF = 0 \qquad (3.14)$$

Which is in fact an over determined set of linear equations and can be simply solved using various numerical methods including Gauss-Jordan or SVD which has better noise cancellation abilities. It is important to consider the calibration step to be a noisy process. Which means it is almost impossible to obtain 100% accuracy in the fundamental matrix. This has severe repercussions when it comes down to the stereo matching process. This means that the epipolar line can not be relied upon completely. This makes the stereo matching more difficult and this uncertainty in the epipolar geometry has been incorporated in the HNN matching technique which is presented in

chapter six. Note that the work in this thesis is different from many other works in the stereo field. Most researchers work on rectified image sets where epipolar lines are in fact scanlines and where the researcher can focus on the actual stereo matching without having to take into consideration the uncertainties of the two-camera geometry. This will in fact decouple the matching from calibration and epipolar line estimation. Therefore this thesis better resembles a real life Machine Vision Application where such assumptions about two-camera geometry can not be used. Note that if we could obtain 100% accurate epipolar lines the matching algorithm would have been considerably different (and simpler).

In order to see what the fundamental matrix actually represents lets look at a real example. Figure 3.10 demonstrated the concept of epipolar geometry and its uncertainties. In this example a point has been randomly selected in the left image and the epipolar line corresponding to this point has been drawn in the other image using software which computes epipolar lines. Once this is established the stereo matching algorithm will travel along this line looking or the most likely candidate using a simple correlation matching method. If the fundamental matrix had been correct, which it is to a high degree as can be seen from the image, the stereo matcher would not have missed. It can be seen that even a one-pixel error in the calculation of the fundamental matrix can mislead a naïve stereo matcher which relies on the epipolar geometry.

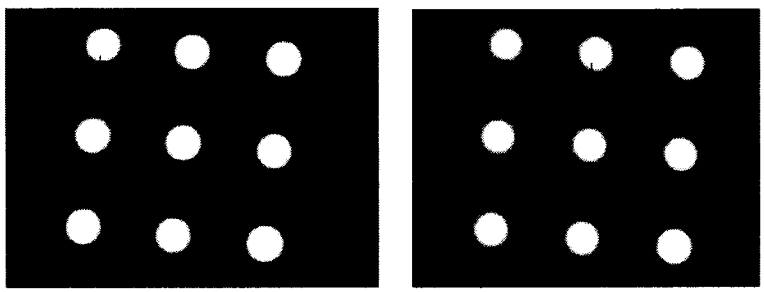

Figure 3.10 Uncertainties in the Fundamental Matrix.

Therefore it is important to use the fundamental matrix effectively, but not rely on it as being 100% accurate.

It is however possible to achieve higher accuracy that reached in this project. It is possible to achieve higher accuracy in both the fundamental matrix computation and projection matrix computation by using third-party calibration tools which make use of both linear and nonlinear optimization tools for finding a numerical solution to the calibration parameters. Most notable is the OpenCV [15] library which is an open source package that is widely used in the computer vision community. Using such packages will enable the researcher to solve various problems and concentrate on the area of research which he/she wishes to focus on. However, this package was not used in out algorithm and a custom calibration routine was written using SVD.

3.3.2 Output of the Calibration

Finally before moving on to the next chapter, a summary of the output of the calibration module is presented:

- Left and right Fundamental Matrices: used for finding match neighborhoods (epipolar geometry).
- Left and right Projection Matrices: used for metric measurement.
- Left and right camera centers: sanity check of the calibration stage, gives the 3D location of the two cameras.
- Error analysis of the calculated projection matrices: determining if the calibration data is useful.

Chapter 4: Hopfield Neural Networks

4.1 Introduction

It is important to introduce Neural Networks before presenting stereo matching since the Hopfield Neural Networks (HNN) will be the tool used in achieving stereo correspondence. Therefore a brief introduction to Neural Networks in general is presented followed by detailed discussion of the HNN which are a special type of Neural Networks. Afterwards, the use of HNN for performing combinatorial optimization will be discussed. There will be little mentioning of stereo matching since the goal of this chapter is to introduce HNN and give an overview of how it could be used in solving optimization problems. The next chapter on stereo matching will present the problem of correspondence and will also discuss how the stereo matching problem can be viewed as a combinatorial optimization problem. There are many ways of solving stereo matching, and recently combinatorial optimization techniques have gained much popularity [16, 17].

What the reader should get from this chapter is a basic understanding of Neural Networks, an understanding of how HNNs work and how they can be used for solving combinatorial optimizations. The following is an introduction to Neural Networks.

Artificial neural networks (ANNs) go by many names such as connectionist models; parallel distributed processing models, neuro-morphical systems, self-organizing systems and adaptive systems. ANNs are composed of simple elements operating in parallel. These elements are inspired by the biological nervous systems. An ANN is an information processing structure that tries to imitate human abilities in perceptron, vision, associative memory and pattern recognition. ANNs are being developed as a technological discipline that can automatically develop operational capabilities to adaptively respond to information environment. An ANN is either hardware or a computer program that strives to simulate the information processing capabilities of its biological example. ANNs are typically composed of a great number of interconnected artificial neurons, which are simplified models of their biological counterparts.

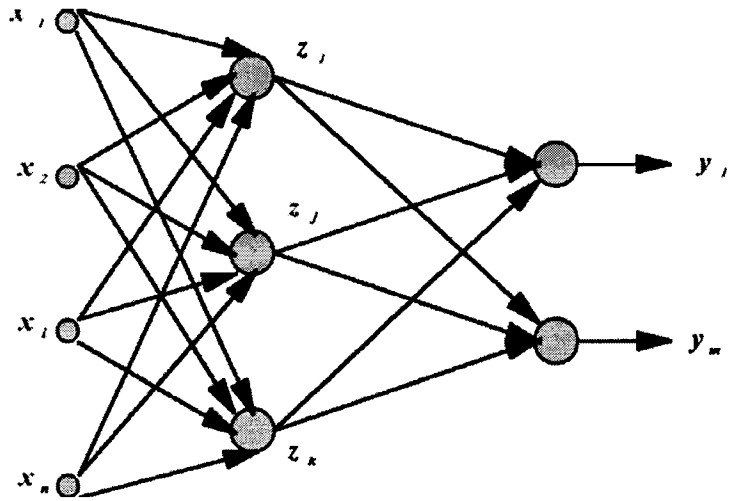

Figure 4.1 Structure of a typical Neural Network

ANNs acquire knowledge through learning and store this knowledge within the inter-neuron connection strengths known as weights. ANNs provide an analytical alternative to conventional techniques, which are often limited by strict assumptions of normality, linearity, variable independence etc. The true power of ANNs lies in their ability to represent both linear and non-linear systems and in their ability to learn the relationships directly from the data being modeled. Figure 4.1 shows a typical NN. Note the fact that a typical NN is simply a network of various "neurons". These neurons can be thought of as processing elements and they closely resemble the concept of biological neurons. They resemble the neurons in the human brain in various ways. For example, both have "states" as in they are either off or on. Also the concept of synaptic weights is common in both models. Therefore it is essential to first understand these processing elements or neuron models before discussing neural networks. The next section provides an overview of what these neurons are and the different mathematical models used to represent them.

4.1.1 Neural Network Training

Before moving on to the next section, it is vital to discuss the dynamic behaviors in NNs, in other words the idea of "training" in NNs. Commonly, ANNs are adjusted or trained so that a particular input leads to a specific desired or target output. Figure 4.2 shows the block diagram for a supervised learning ANN, where the network is adjusted based on comparing the neural network (NN) output to the desired output until the network output matches the desired output. After the network is trained, the network can be used to test new input data using the weights provided from the training session, the input data is fed through a NN structure to get an output.

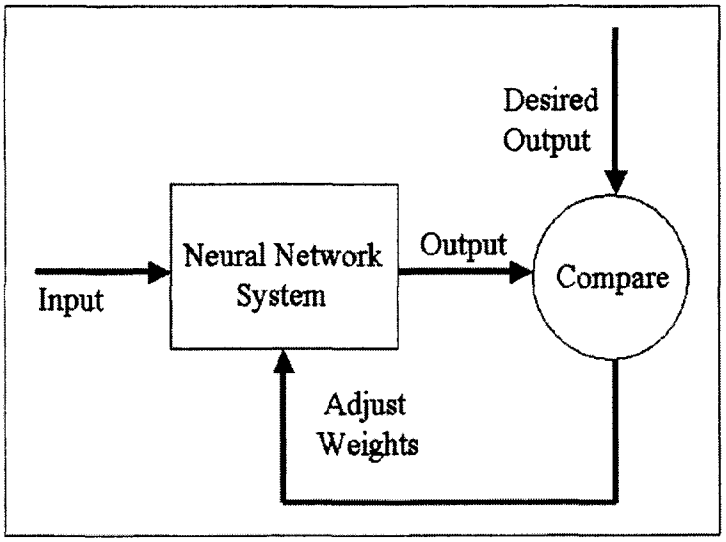

Figure 4.2 Learning Ability of ANNs

4.1.2 Advantages of Neural Networks

The reason why NNs have been chosen in this work will be presented herein. Going through the stereo matching literature one will notice a very obvious absence of Neural Network methods in solving the stereo correspondence problem. However there are many reasons why researchers have a great interest in Neural Networks and which make them attractive for solving various problems, from classification to optimization.

The major advantages and disadvantages of Neural Networks (NNs) NNs in modeling applications are as follows:

- NNs have high tolerance to noisy data.
- NNs have the ability to model multi-dimensional nonlinear relationships.
- Neural models are simple and the model computation is fast.
- Parallel implementation is easy.
- NNs Learn from example, are capable of generalizing data, which makes it possible to process new, imperfect and distorted data.
- There is no need to assume an underlying data distribution such as usually is done in statistical models.
- It is easier to update neural models whenever device or component technology changes.
- NNs can handle different kinds of environments such as dynamic and complex.
- NNs have the ability to implicitly detect complex nonlinear relationships between dependent and independent variables.

The most important motivation behind using NNs in this thesis is the fourth item, the fact that parallel processing is a definite possibility with a NN solution. This will be more thoroughly explained later on.

ANNs have been successfully applied to broad spectrum of applications. For more information regarding the applications of NNs see Haykin [4], which gives an excellent overview on Neural Networks and its applications. Four different areas of research that

are of great significance will are: pattern classification (most popular), data mining, prediction and association (HNN).

4.2 Artificial Neuron Model

As mentioned before, the Neurons are the building blocks of neural networks and as such they merit some attention. This section briefly described these processing elements which will later be used in creating large neural networks.

Similar concepts of the biological neuron are applied in the artificial neuron, in other words the idea of the artificial neuron is inspired by its biological counterpart. But still the artificial neuron is by far a simplified model compared to the biological neuron. In its simplest form, as shown in Figure 4.3, a neuron sums a set of input, weighted by the strength of its connections (synaptic weights) and takes the overall value and uses it as an input to some internal function (activation function) and outputs the resulting value of this function through a set of weights to other neurons in the network.

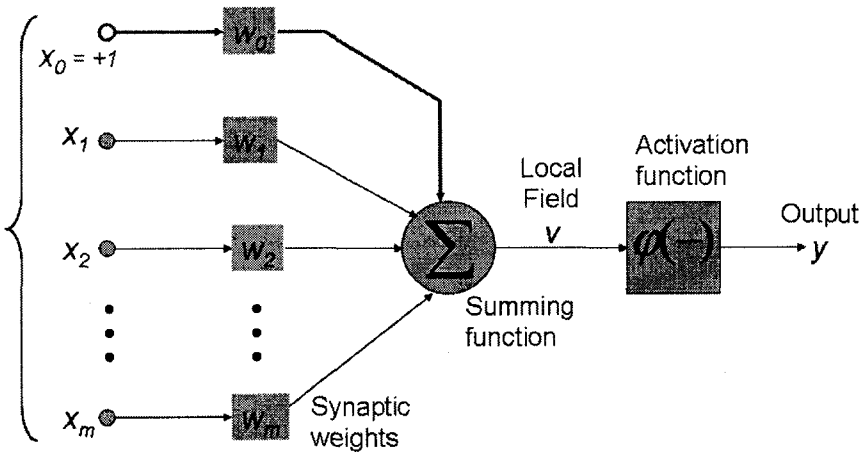

Figure 4.3 A general artificial neuron

In general, an artificial neuron contains four components: input, weights, processing unit and output. The input is obtained from the other neurons or from the outside of the neural network. The weights (w_1, w_2 . . . w_i) are connections between

neurons which have an influence on the response of the neurons. A processing unit sums up the weighted input and threshold value and passes the signal through the activation function (transfer function) to the output. The output connection carries the output (φ) to the other neurons or outside of the neural network. During the processing of the information the input is multiplied by the appropriate weight and then passed through the transfer function, after summing up of all the weighted input and threshold value. Most Artificial Neurons are similar in this way and differ only in their "Activation" function. That is most Artificial Neurons act as small processing elements, connected to other, similar elements in a large parallel processing network. But the way they operate on their internal input or their activation function is what sets them apart.

4.2.1 Activation Functions

The activation function is a function that is used to transform the activation level of an input neuron into an output signal. Typically, activation functions have a squashing effect. The activation function gives information to the processing neuron about how active is the connections to this neuron. Different types of activation functions exist, but in all these cases their purpose is to determine the neuron's output signal level as a function of the input signal level to the neuron. Several models have been shown in Figure 4.4. Keep in mind that the discreet HNN which will be discussed in the next sections uses the threshold function shown below, and the continuous HNN which is used in the final implementation uses the Sigmoid version.

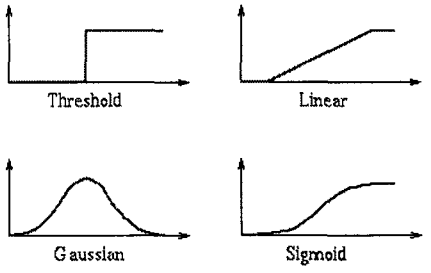

Figure 4.4 Various models of activation functions.

The continuous version usually uses a function of the form:

$$\varphi(v) = \tanh(\frac{\alpha v}{2}) \tag{4.1}$$

whereas the discrete neurons usually follow an equation as shown below:

$$\varphi(v) = \begin{cases} a & \text{if } v < c \\ b & \text{if } v > c \end{cases} \tag{4.2}$$

Note in the case of the HNN, since the network has a time dimension, the equation (4.2) takes on a different form. In that case, if v=c, the value of the function will retain the value at time instant t-1.

4.3 Types of Neural Networks

This section presents three major Neural Network models, Multi-Layer Perceptron; the Self Organizing Maps (SOM) and the Hopfield Neural Networks. There are many other types of NNs; however these three capture the diversity of the various NN models and serves as a good introduction.

4.3.1 Multi-Layer Perceptron

First the Multi-Layer Perceptron (MLP) is the most common NN model. MLP is a hierarchical structure of several perceptron, which uses supervised training methods to train the NN. The training of such a network with hidden layers is more complicated than a single perceptron which does not contain any hidden layers. This is because when there exists an output error, it is hard to know how much error comes from the input node, how much from other nodes and how to adjust the weights according to their respective contributions to the output layer. The problem can only be solved by finding the effect of all the weights in the network. This is solved by using the back-propagation algorithm [4] which is a generalization of the least-mean-square (LMS) algorithm. The back-propagation algorithm uses an iterative gradient technique to minimize the mean-

square-error between the desired output and the actual output of the MLP. The training procedure is initialized by selecting small random weights and internal thresholds. The training data are repeatedly presented to the network and the weights are adjusted until they stabilize which means the mean-square error is reduced to an acceptable value. The whole training sequence involves forward phase and backward phase.

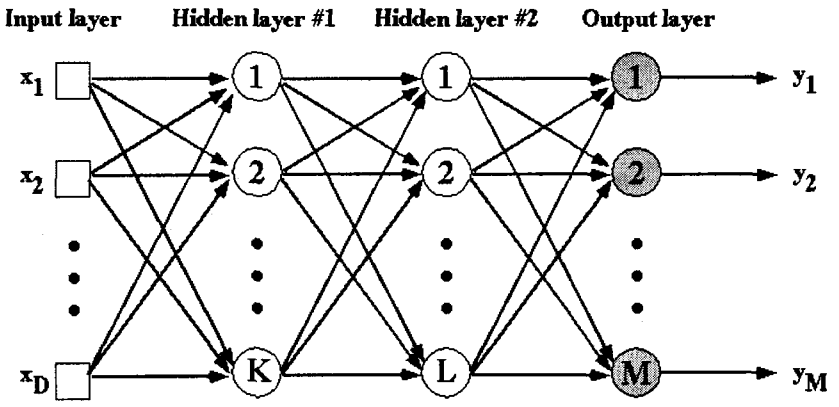

Figure 4.5 Multilayer Perceptron (MLP)

The forward phase estimates the error and the backward phase modifies the weights to decrease the error. The MLP is perhaps one of the most commonly used NNs and they serve as a standard tool for performing classification.

4.3.2 Self Organizing Maps

Second is the Self Organizing Maps (SOMs) which are a data visualization technique invented by Professor T. Kohonen [18] which reduce the dimensions of data through the use of self-organizing neural networks. The problem that data visualization attempts to solve is that humans simply cannot visualize high dimensional data and this tool is created to help us understand this high dimensional data. The way SOMs go about reducing dimensions is by producing a map of usually 1 or 2 dimensions which plot the similarities of the data by grouping similar data items together. So SOMs accomplish two things, they reduce dimensions and display similarities. Just to give you an idea of what a

SOM looks like, Figure 4.6 shows an example of a SOM. As you can see, like colors are grouped together such as the greens are all in the upper left hand corner and the purples are all grouped around the lower right and right hand side.

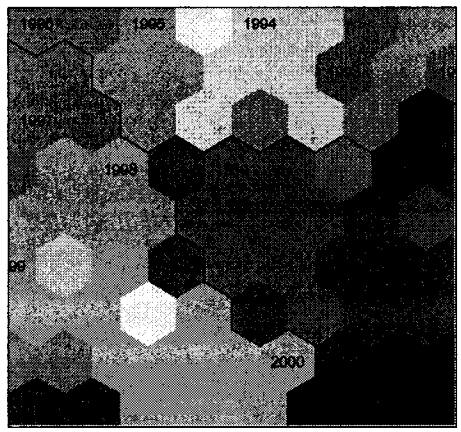

Figure 4.6 Self Organizing Maps

The first step in constructing a SOM is to initialize the weight vectors. From there you select a sample vector randomly and search the map of weight vectors to find which weight best represents that sample. Since each weight vector has a location, it also has neighboring weights that are close to it. The weight that is chosen is rewarded by being able to become more like that randomly selected sample vector. In addition to this reward, the neighbors of that weight are also rewarded by being able to become more like the chosen sample vector.

4.3.3 Hopfield Network Model

The third class of Neural Networks that will be presented here as a survey is the Hopfield Neural Networks (HNNs). Although they are presented here as merely another class of Neural Networks, they are the main tool with which we have implemented in a machine vision algorithm and thus the next section will present a full detail discussion of this class. However for the sake of completion a brief explanation is presented here.

In the beginning of the 1980s Hopfield published two scientific papers [19, 20], which attracted much interest. This was the starting point of the new era of neural networks, which continues today. Hopfield showed that models of physical systems could be used to solve computational problems. Such systems could be implemented in hardware by combining standard components such as capacitors and resistors. The Hopfield neural network is a simple artificial network which is able to store certain memories or patterns in a manner rather similar to the brain - the full pattern can be recovered if the network is presented with only partial information. Note that unlike most other NNs which are used for classification purposes, the HNN is used mainly as a memory system, and in this case, as an optimization tool. Furthermore there is a degree of stability in the system - if just a few of the connections between nodes (neurons) are severed, the recalled memory is not too badly corrupted - the network can respond with a "best guess" which makes this algorithm immune to noisy data. Of course, a similar phenomenon is observed with the human brain, for instance during an average lifetime many neurons will die but we do not suffer a catastrophic loss of memory. The nodes in the network are vast simplifications of real neurons - they can only exist in one of two possible "states" - firing or not firing (for binary HNNs, for continuous HNNs there are infinitely many states). Every node is connected to every other node with some strength as shown in Figure 4.7. At any instant of time a node will change its state (i.e. start or stop firing) depending on the inputs it receives from the other nodes. If we start the system off with any general pattern of firing and non-firing nodes then this pattern will in general evolve with time and converge to a stable configuration (more on this will be presented). One might imagine that the firing pattern of the network would change in a complicated perhaps random way with time. The crucial property of the HNNs which renders it useful for simulating memory recall is the following: we are guaranteed that the pattern will settle down after a long enough time to some fixed pattern. Certain nodes will be always "on" and others "off".

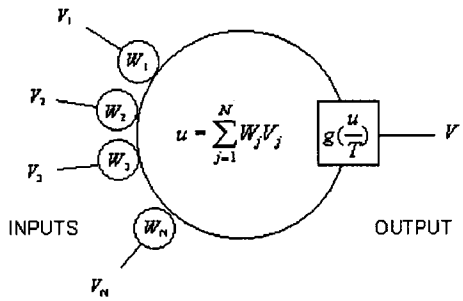

Figure 4.7 A node, or a neuron in an HNN

Furthermore, it is possible to arrange that these stable firing patterns of the network correspond to the desired memories we wish to store if the goal is to use the HNN as a content addressable memory. However if optimization is the tool we can use the evolution of the network (the network minimizes its energy) to minimize a complicated multidimensional energy function which represents out problem. This is shown in Figure 4.8 where the energy of the HNN is shows to be decreasing and converging to a stable state.

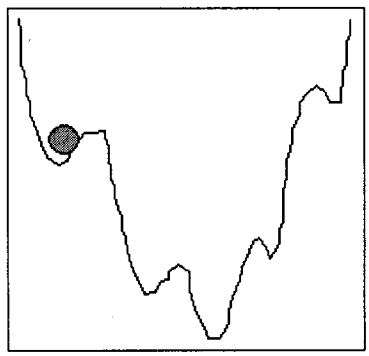

Figure 4.8 The HNN moves along an n-dimensional energy surface

The reason for this will be thoroughly explained in the next section and mathematics of HNN will be discussed to model the behavior of the HNN, but for now lets take a lighter approach to discussing the evolution of the HNN using analogy.

Imagine a ball rolling on some bumpy surface as in Figure 4.8; we imagine the position of the ball at any instant to represent the activity of the nodes in the network. Memories will be represented by special patterns of node activity corresponding to wells in the surface. Thus, if the ball is let go, it will execute some complicated motion but we are certain that eventually it will end up in one of the wells of the surface (a minima). We can think of the height of the surface as representing the energy of the ball. We know that the ball will seek to minimize its energy by seeking out the lowest spots on the surface -- the wells. Furthermore, the well it ends up in will usually be the one it started off closest to (this is where proper initialization of the HNN is important).

This is the equivalent of gradient descent, but in the context of binary problems. This is why HNNs are called combinatorial optimization tools. Because as opposed to, for instance the Newton-Raphson algorithm, they are designed to solve discrete optimization problems.

4.4 Hopfield Neural Networks

As mentioned earlier in the beginning of the 1980s, Hopfield [19, 20] and his colleges published two scientific papers on neuron computation. Hopfield showed that highly interconnected networks of nonlinear analog neurons are extremely effective in solving optimization problems. In fact, the Hopfield Neural Networks have a rather easy VLSI implementation. Hopfield's original schematic for the Hopfield Neural Network was actually an analog circuit implementation as shown in Figure 4.9.

From that time on, people has being applying the Hopfield network to solve a wide class of combinatorial optimization problems. In a discrete-time version, the Hopfield network implements local search. In a continuous-time version, it implemented gradient decent. The neural network contains only one layer. The input feeds to the layer at time zero and then the feedback from the output nodes are used as input. Output nodes are fed back to the inputs via variable connection weights. There are no self-feedback connection weights in this NN. Figure 4.10 shows the connections and the layers of the HNN.

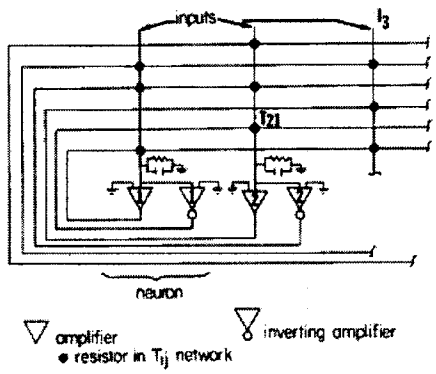

Figure 4.9 The original circuit layout of the Hopfield Neural Network

When used for pattern association tasks, the training method for the Hopfield NN is a supervised training method. The HNN knows the input and the output values of the HNN before the training process. The only unknown for this HNN is weights. In fact, this is the case regardless of the application that the HNN will be used in. Regardless of the mode of the operation, the weights are the unknowns which have to be found and tailored to the given problem. The discrete HNN takes the binary input values +1 and -1, whereas the continuous HNN which have been used in this thesis use a continuous range of values from 0-1.

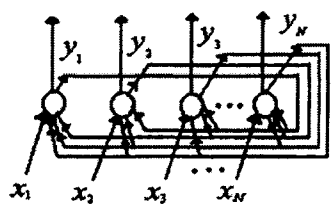

Figure 4.10 HNN layout

4.4.1 Hebb's Trainig Rule

The training of this network is done by the Hebb rule when pattern association is the goal. The training process is given below:

$$\begin{cases} w_{ij} = \sum_{s=0}^{M-1} x_i^s x_j^s, & i \neq j \\ w_{ij} = 0 & i = j \end{cases} \qquad (4.3)$$
$$where \quad 0 \leq i, j < N$$

Where W_{ij} is the weight connection between the input node i and the output node j, X^s_i is the input element of training pattern s, M is the number of patterns used for training and N is the number of nodes in the layer (number of input components). When using the HNN for pattern association, during the recognition process, an unknown input is applied to the NN at time zero. Then the NN forces the output to match one of the trained patterns. The recognition process is shown below:

$$y_j(0) = x_j, \qquad 0 \leq j < N$$
$$y_j(t+1) = f_h\left[\sum_{i=0}^{N-1} w_{ij} y_j(t)\right], \quad 0 \leq j < N \qquad (4.4)$$

Where y_j is the output of the output node j, w_{ij} is the connection weight from the output of the output node i to the input of the output node j, N is the number of input elements and the f_h is the hard limiting nonlinearity activation function. The activation function in this case could be a step function, which goes from -1 to +1 at the. The recognition process continues until there is no change in the output states on successive iterations. Of course the network has to be left to evolve for a number of iterations (order of hundreds, depending on the size) to ensure the network is in a stable point.

4.4.2 Energy Function of the HNN

Note that the most important aspect of the HNN is the energy function which explains its convergent behavior. This behavior is the means with which pattern

association and optimization is accomplished in HNNs. In fact it was Hopfield [20] who first showed in his work that this can be accomplished. The next section explains the process of optimization using the HNNs.

4.5 Hopfield Neural Network Approach to Combinatorial Optimization

A large class of logical problems arising from real world situations can be formulated as optimization problems, and thus qualitatively described as a search for the best solution. These problems are found in engineering and commerce, and in perceptual problems which must be rapidly solved by the nervous systems of animals. Well-studied problems from commerce and engineering include: Given a map and the problem of driving between two points, which is the best route? Given a circuit board on which to put chips, what is the best way to locate the chips for a good wiring layout? Analogous, but only partially characterized problems in biological perception and robotics include: given a monocular picture, what is the best three-dimensional description of the locations of the objects? Indeed, what are the "objects"? In each of these optimization problems, an attempt can be made to quantify the vague criterion "best" by the use of a specific mathematical function to be minimized.

4.5.1 Computational Power of a Connectionist Model

In addition, the stereo matching problem that we need to solve in this research can also be viewed as a combinatorial optimization task. This will be discussed in more details in the next chapter. But one might wonder why Hopfield decided to use a "neural network" model to try to solve the classical optimization problem. The reason is that he realized that the computational powers routinely used by nervous systems to solve perceptual problems must be truly immense, given the massive amount of sensory data continuously being processed, the inherent difficulty of the recognition tasks to be solved, and the short time (msec-secs) in which answers must be found. Also he noted

that an understanding of biological computation may also lead to solutions for related problems in robotics and data processing using non-biological hardware and software. It is clear from studies in anatomy, neurophysiology, and psychophysics that part of the answer to how nervous systems provide computational power and speed is through parallel processing. The mammalian visual system computes elementary feature recognition massively in parallel. At the level of neural architecture, anatomy and neurophysiology have revealed that the broad category of parallel organization is manifest in several different but interrelated forms. Parallel sensory input channels, such as the individual rods and cones in the vertebrate retina, allow rapid remote sensing of the environment and data transmission to processing centers. Likewise, parallel output channels, for example in corticocortical projections in the cortex, connect different processing modules. Another manifestation of parallelism occurs in the large degree of feedback and interconnectivity in the "local circuitry" of specific processing areas. The idea that this large degree of local connectivity between the simple processing units (neurons) in a specific processing area of the nervous system is an important contribution to it's computational power has led to the study of the general properties of neural networks and also several "connectionist" theories in perception. The connectionist theories employ logical networks of two-state neurons in a digital clocked computational framework to solve model pattern recognition problems.

In essence, seeing the power of a connectionist computational model in human beings, Hopfield decided to harness the power of highly interconnected processing element for the purpose of optimization. And he did so in fact solve the classical Traveling Salesman Problem (TSP) in his paper [20] and demonstrated the feasibility of the solution that can be obtained using his HNN as shown in Figure 4.11. The general structure of the analog computational networks which can solve optimization problems is shown in Figure 4.9. These networks have the three major forms of parallel organization found in neural systems: parallel input channels, parallel output channels, and a large amount of interconnectivity between the neural processing elements. The processing elements, or "neurons", are modeled as amplifiers in conjunction with feedback circuits comprised of wires, resistors and capacitors organized so as to model the most basic computational features of neurons, namely axons, dendritic arborization,

and synapses connecting the different neurons. The amplifiers have sigmoid monotonic input-output relations, as shown in Figure 4.4.

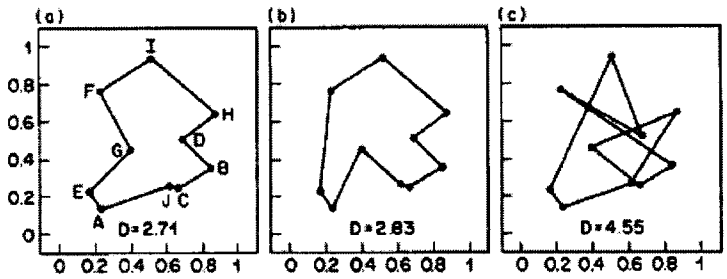

Figure 4.11 Solutions to the traveling salesman problem found using continues HNN in a, b and discrete HNN in c.

4.5.2 Evolution of the HNN

Although this "neural" computational circuit is described here in terms of amplifiers, resistors, capacitors, etc., we have shown that networks of neurons whose output consists of action potentials and with connections modeled after biological excitatory and inhibitory synapses could compute in a similar fashion to this conventional electronic hardware. We can describe the discrete Hopfield net in discrete time as:

$$y_i(n+1) = \text{sgn}\left(\sum_{j=1}^{N} w_{ij} y_j(n) - b_i + x_i(n) \right) \qquad i = 1,...N \tag{4.5}$$

Where sgn represents the threshold nonlinearity (0,1) and b is a bias. We assume that the update is done sequentially by neuron number; however there are other methods of updating such as random updating where neurons are updated at random. The input is a binary pattern x = [x_1, x_2, ... x_N] and works as an initial condition; it is presented to the network and then taken away to let the network relax so it disappears from equation(4.5). Due to the sgn function we can see that the points visited during relaxation are the vertices of a hypercube in N dimensions (if the network is a discrete model).

We presented the practical aspects of Hopfield networks, but we assumed that the dynamics do in fact converge to a point attractor. Under what conditions can this convergence be guaranteed? This is a nontrivial question due to the recurrent and nonlinear nature of the network. The importance of the Hopfield network comes from a very inspiring interpretation of its function provided by Hopfield. Due to the extensive interconnectivity, it may seem hopeless to try to understand what the network is doing when an input is applied [21]. We can write the dynamic equations for each neuron, but due to the fact that these different equations are highly coupled and nonlinear, their solution seems to be beyond our reach. In practice this is not so. When the weight matrix is symmetric, the neurons are threshold nonlinearities and the biases are zero, we can show that the network accepts an energy function E:

$$E = -\frac{1}{2}\sum_i \sum_j w_{ij} y_i y_j \tag{4.6}$$

This is the most important behavior of an HNN. Note that the energy function here assumes a one-dimensional HNN and zero biases. However, the HNN used in this thesis is a two dimensional HNN with non-zero biases, the energy function of this HNN will be described later on. The energy function is a function of the configuration of the states [21] that is *non-increasing* when the network responds to ANY input. This is where the HNN proves its use a minimization tool since any unction that is rearranged in the form of the energy function can be minimized using the HNN. We can show this by proving that every time one neuron changes state, E decreases. When the neuron does not change, E remains the same. This means that the global network dynamics pulls the system state to a minimum (along the gradient of E), that corresponds to one of the minima of the system. The location of the minimum in the input space is specified by the weights chosen for the network. Note that when an optimization is carried out using the HNN these weights are the parameters that need to be adjusted and that there are no patterns given to the system, except a random initialization pattern used to start the system. Once the system reaches the minimum it will stay there, so this minimum is a fixed point, or an attractor. When the Hopfield network receives an input, the system state is placed somewhere in weight space. The system dynamics then relax to the

memory that is closest to the input pattern. Around each fixed point there is thus a basin of attraction that leads the dynamics to the minimum. This explains why the Hopfield network is so robust to imprecisions (added noise or partial input) of the input patterns. Once the system state is in the basin of attraction for a stored pattern, the system relaxes to the undistorted pattern as shown in Figure 4.12 .

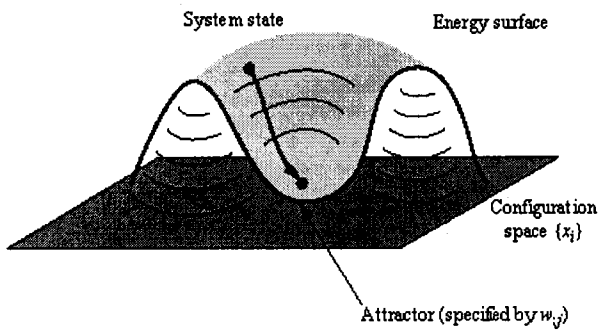

Figure 4.12 Energy surface of an HNN

There are some practical problems, however, since when we load the system with patterns, spurious memories are created which may attract the system to unknown positions. An energy surface with minima and a basin of attraction creates the mental picture of a computational energy landscape, which is similar to a particle in a gravitational field. This metaphor is very powerful, because suddenly we are talking about global properties of a tremendously complex network in very simple terms. We have ways to specify each connection locally, but we also have this powerful picture of the computational energy of the system. Hopfield networks can be used in optimization, because we can link the energy surface to problem constraints. The optimal solution is found by relaxing the system with the present input to find the closest solution (the attractor). Mapping the problem solution to the network is the difficult part and has to be done on a case-by-case basis, and in the case of our stereo vision problem this encoding of the problem with the HNN energy function will be discussed in chapter six.

Chapter 5: Stereo Correspondence

5.1 Introduction

Image correspondence can be defined as a mapping between two images, both spatially and with respect to intensity. If the two images are denoted by I_1 and I_2 where $I_1(x, y)$ and $I_2(x', y')$ are image coordinates which map to the intensity value of the corresponding pixel, then the matching between these two images can be expressed as:

$$I_2(x', y') = g(I_1(f(x, y))) \tag{5.1}$$

Where g is a 2D intensity transformation and f is a 2D spatial-coordinate transformation. g should be considered when two images are taken with different types of sensors; it is not necessary in a typical stereo vision system with only CCD cameras. Therefore, matching is to find a transformation f that maps spatial coordinates x and y, to new spatial coordinates x' and y' as shown below.

$$(x', y') = f(x, y) \tag{5.2}$$

Some typical geometric transformation between the two images is shown in Figure 5.1.

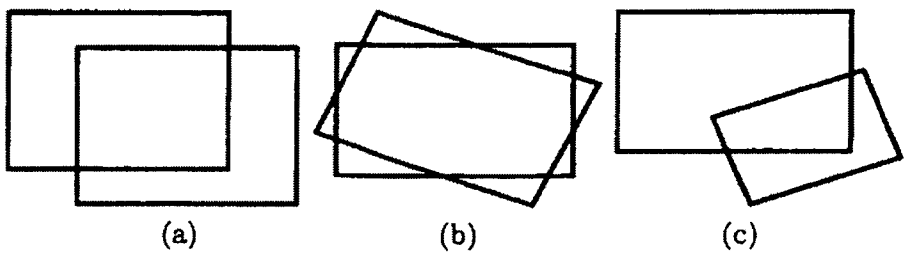

Figure 5.1 Typical geometry transformation: (a) translation. (b) Rotation. (c) Rigid transformation.

Translation transformation occurs when the images are misaligned by a small shift due to a change in the camera's position. Rotation transformation is caused by a camera rotation around the axis. Rigid transformations are those where the objects in the images retain their relative size and shape. If the transformation between two images is 2D, the spatial transformation f can be expressed by a single equation that maps each point in the first image to a new location in the second image. However, because the 3D-2D perspective projection is an irreversible one, it is impossible to find a spatial mapping function between a pair of stereo images. Therefore, stereo matching is defined as locating a pair of image points resulting from the projection of the same object point by some similarity constraints of the pixel colors. The relation can be written as follows:

$$I_2(x', y') = I_1(x + d_x, y + d_y) \tag{5.3}$$

Where d_x and d_y are the location differences of the matching pairs along the x and y coordinates respectively. Note that not all the points in one image can find their corresponding points in the other image; this is called occlusion. It is because a given object point may not have a projection in both images. For instance a point may appear in the left image but not in the right image because it becomes hidden due to the position and orientation of the right camera.

5.2 Disparity and Disparity Map

Disparity specifies the offset of a pixel in the first image to its match in the second image. Without loss of generality, we can assume that the two image planes are parallel. As shown in Figure 5.2, an object point P projects in the left image on p(x, y) and in right image on p'(x', y'). Since the two image planes are parallel, we have y = y'. Therefore the disparity d of P on this pair of images is one dimensional, and d = x'-x. Note that in our case this was not true. This project was carried out using two cameras with arbitrary geometry therefore the disparities were two dimensional. However recently most stereo matching research is carried out on parallel geometry since changing the non-coplanar camera geometry to a coplanar geometry is a matter of rectifying the images [9]. This is

done by calibrating the cameras and projecting the non-planar images to a coplanar set of images.

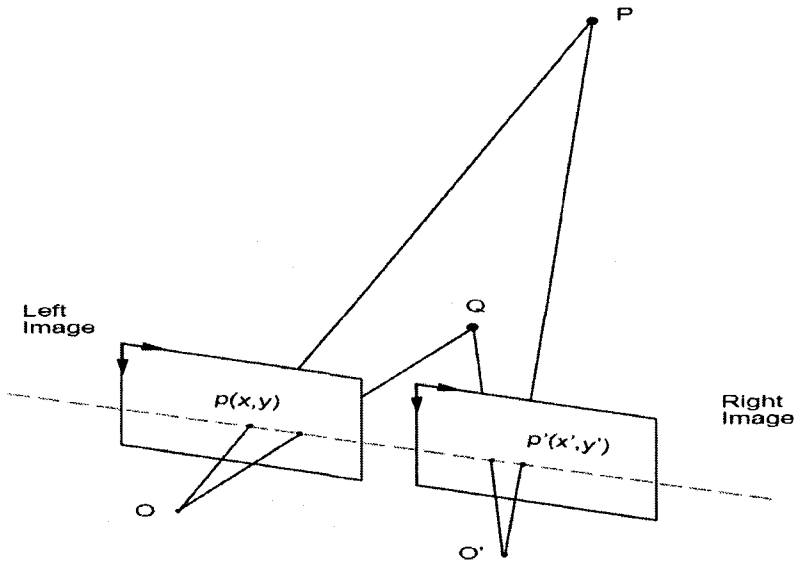

Figure 5.2 Geometric depiction of disparity

Matching results can be stored in a disparity map, while the result of dense matching is recorded in a dense disparity map. A disparity map is formed by the disparities of all the matched points and can be displayed as an image. It is defined as an integer valued array where each entry stores the disparity value of the same image location. For example, if the disparity map D records the matching result between left and right images, given a pair of matched points (p(x, y), p' (x', y')) , then D(x, y) = k with k = x' - x. In that case, k is considered as the intensity value (gray value) of the pixel at (x, y) of the disparity map picture. This is a very easy way of visualizing the disparities and a convenient way to inspect the quality of a stereo matching. An example of a dense disparity map is shown in Figure 5.3. This is a particularly accurate disparity map since the surface is very smooth and the change of gray level corresponds to the change in the elevation of the object. The fact is that the disparity value indicates the distance from the cameras to the object point. If an object point is infinitely far away, then its projection

onto the two image planes will be at the same location, and the disparity will be zero. If an object is close to the cameras then the disparity will be large. Disparity is inversely proportional to the distance between an object and the camera system.

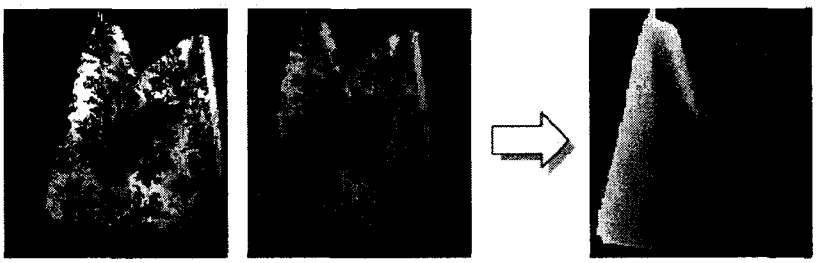

Figure 5.3 Example of a disparity map

Now the reader might realize why a robot vision system might need a disparity map of the environment, to get information regarding the depth of objects. In fact the stereo matching algorithm which will be developed in the next section is the means with which the robot is able to estimate the shape and depth of objects that have been placed within the field of view of the cameras. To better see how depth and disparity are inversely proportional see Figure 5.4.

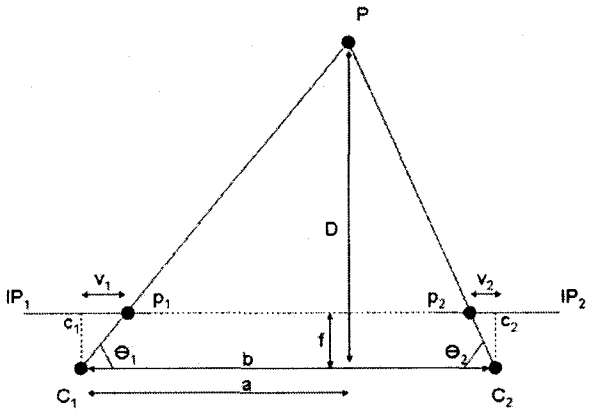

Figure 5.4 Relationship between depth and disparity.

From the above figure it can be seen that given the image location of a point, if the corresponding point in the other image found, we can find the depth of the physical location that corresponds to these points using the formula:

$$D = \frac{bf}{v_1 + v_2} \tag{5.4}$$

Where D represents depth from camera, b is the baseline (distance between the two camera centers) and v_1 and v_2 are the image locations.

5.2.1 Sparse Disparity Maps

Going back to Figure 5.3, note that in most machine vision applications a dense map is not required. That is we only need to match the pixels at certain, "critical" points. These points are referred to as features and are usual areas of high contrast such as edges or corners. The method for finding these features or the feature extraction technique is used in the next chapter. But note that the number of points that need to be matched in a typical machine vision application is less than that of a dense map where every single point is matched. This makes things a bit easier for two reasons, one is that obviously fewer points need to be found and second that the most challenging part of a matching is to find correspondence for points that have low texture. But as we explained, feature points usually have high contrast and texture which means they can be found with a much higher degree of certainty than those points belonging to even and textureless areas. In brief, we require a sparse depth map rather than a dense depth map.

5.2.2 Features

Sparse matching matches feature points in an image as explained. Since these feature points arc easily distinguishable from other points, they are also easier to match. Sparse matching can be divided into two steps:

1. Identifying the interest points (feature extraction)

2. Matching between these sets of points (correspondence)

Identifying feature points in an image involves finding interest points such as corners, junctions or dots. Figure 5.5 shows examples of different neighborhood configurations around an interest point.

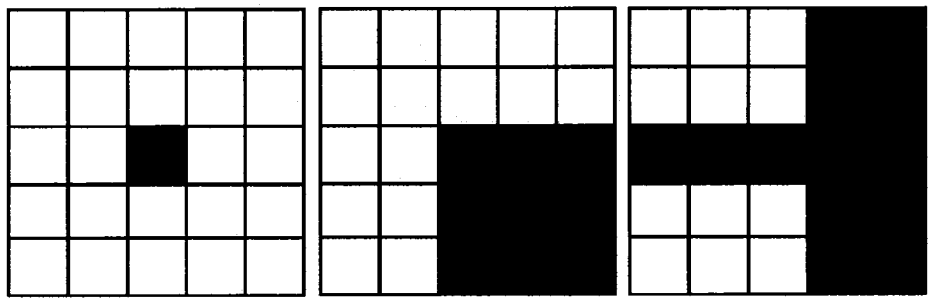

Figure 5.5 Example Patterns for interest points in a 5 x 5 neighborhood: a) dot, b) corner, c) junction

More on feature detection and the exact method used for the implementation of this work will be discussed in the next chapter. For now we will discuss stereo constraints which play a central role in stereo matching. These are constraints on how a point can be matched to other points and a starting point for any stereo matching algorithm.

5.3 Stereo Matching Constraints

Stereo matching process is a very difficult search procedure. In order to minimum false matches, some matching constraints must be imposed. Below is a list of the commonly used constraints.

5.3.1 Epipolar Constraint

Perhaps the most useful stereo constraint is the epipolar constraint which has been fully used in this thesis. Given a point p in the left image, the object point P that was projected on p may lie anywhere on the ray defined by O and p (keep in mind that it is impossible to recover the depth of a point from a single image). However, the image of

this ray in the right image is the epipolar line defined by the corresponding point p' and the epipole e'. Therefore, the correct match of p must lie on this corresponding epipolar line in the right image. This constraint is known as the epipolar constraint and is shown in Figure 5.6; it establishes a mapping between points in the left image and lines in the right image and vice versa. If we assume that the epipolar geometry is known, the search for the match of p in the right image can be restricted to the search along the epipolar line of p. The matching problem is reduced from a two-dimensional search to a one-dimensional search. This is a considerable simplification of the problem; for instance, the search over 1000 x 1000 = 10^6 pixels (a 1000 x 1000 image) will be reduced to a search over 1500 pixels(a diagonal line at most). The mathematics of this constraint was discussed previously.

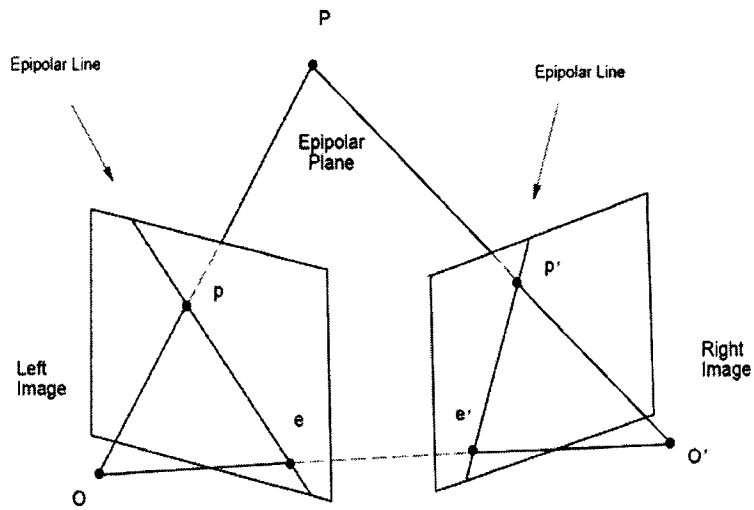

Figure 5.6 The epipolar constraint significantly reduces the search neighborhood.

It was also explained how this constraint is algebraically represented as the fundamental matrix. Also it was shown how the fundamental matrix can be found during the calibration step. In brief, the epipolar constraint can be found if the fundamental matrix is accurately found, this way the algorithm can limit its search neighborhood. But realistically this is impractical since the fundamental matrix could suffer from

inaccuracies. Thus one way to overcome this is to search around epipolar lines, for instance the search can be carried out along three lines around and including the epipolar line.

5.3.2 Ordering Constraint

Another important constraint used in stereo matching is the ordering constraint. The order constraint states that stereo projections always preserve the order of points along the according epipolar line. As shown in Figure 5.7, if point n is on the right side of point m on the epipolar line, then the matching point n' of n must lie on the right side of the matching m' of m. The reason is that it is geometrically impossible for points projected from the same opaque surface to be differently ordered in the stereo image pair.

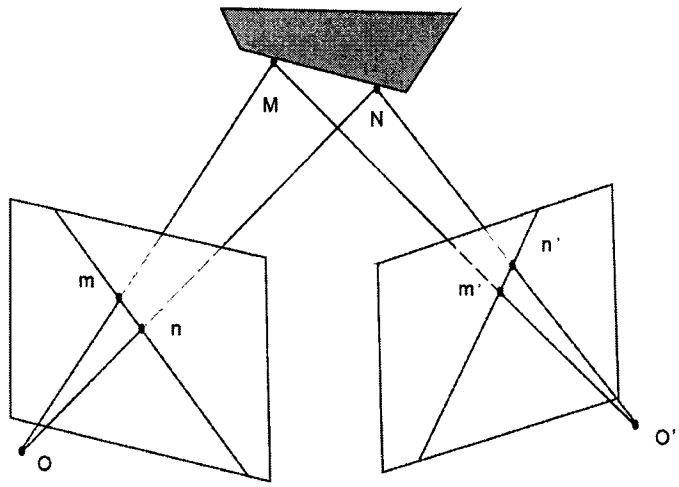

Figure 5.7 Order constraint.

If the ordering constraint is used effectively it could help in further reducing the search neighborhood. As the reader might guess, a good algorithm is one that can make

full use of all stereo constraint. This is because further reducing a search neighborhood means less ambiguity and higher accuracy. However care must be taken when using the ordering constraint since it depends on previously matched points. If this incorrect information is used in order to ascertain the neighborhood using the ordering constraint the algorithm will have no chance of finding the right candidate. There are many ways to overcome this but this is not discussed since the ordering constraint was not used in this thesis. The reason for that was that since an objective function has to be formed, it is difficult to incorporate all matching constraints effectively within this objective function. This constraint has been left out of the matching process. However a future work built upon this thesis could consider including more constraints. Keep in mind that our stereo algorithm uses the Disparity Gradient [22] constraint, which could be a substitute for the ordering constraint. This constraint is discussed later in this section.

5.3.3 Continuity Constraint

The next constraint that is discussed here is the continuity constraint. The continuity constraint is also known as surface smoothness constraint. The underlying idea of this constraint is that the world is mostly made up of objects with smooth surfaces. It states that disparity varies smoothly on object surfaces; sharp changes of disparity occur at object boundaries. In the ground truth disparity map Figure 5.3, we can recognize the shape of the objects in the scene since the sharp intensity change at the object's boundary. This abrupt change indicates the disparity discontinuity at the object boundary. Meanwhile on the surface of the objects intensity changes smoothly and uniformly, which indicates the continuity of disparity values.

5.3.4 Uniqueness Constraint

The other constraint used in this thesis is the unique constraint. The uniqueness constraint states that one image point has at most one match in the other image. It is impossible that one object point can project at more than one location in only one image; while there is no match in the case of occlusion. The uniqueness constraint can simplify the computation and can be used to validate the matching results. Note that this

constraint is explicitly included in our objective function as will be discussed in this next chapter.

5.3.5 Disparity Gradient

Burt and Julesz [23] pointed out that Disparity Gradient dictates binocular fusion when several objects occur near one another in the visual field. They argue that there is a Disparity Gradient limit of approximately 1 for most human subjects in their experiments. Furthermore, order reversal occurs in two potential matches when the Disparity Gradient is allowed to be larger than 2. There have been numerous articles exploiting the Disparity Gradient limit [24]. However, the reason why it is not as popular as the other constraint could be the fact that it was inspired by the human visual system rather than being resulted from mathematical derivation. We have used this constraint in our objective function for two reasons, one is because it is able to contain the ordering constraint, the other is the fact that it fits nicely with the HNN objective function and can be easily included in our optimization as will be discussed in the next chapter.

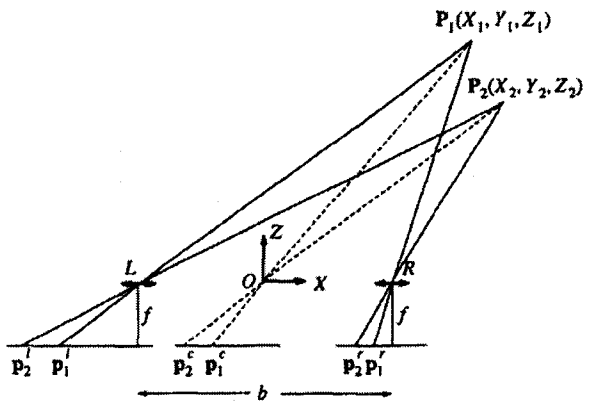

Figure 5.8 Defining Disparity Gradient in stereo vision.

Figure 5.8 depicts the camera geometry for stereo vision where the camera optical axes are parallel to each other and perpendicular to the baseline connecting the two

cameras L and R. For a point P (X, Y, Z) in the 3-D scene, its projections onto the left image and the right image are $p^l(x^l, y^l)$ and $p^r(x^r, y^r)$. Because of this simple camera geometry, $y^l = y^r$ and the disparity d is inversely proportional to the depth Z.

$$d = x^l - x^r = \frac{bf}{Z} \tag{5.5}$$

Where f is the focal length of the camera lens and b is the separation of the two cameras. Given two points $P_1(X, Y, Z)$ and $P_2(X, Y, Z)$, their Disparity Gradient (δd) can be defined as δd = difference in disparities/cyclopean separation, where cyclopean-separation is the average distance between p_1^l, p_2^l and p_1^r, p_2^r.

$$\delta d = 2 \times \frac{\left|\left(x_2^l - x_2^r\right) - \left(x_1^l - x_1^r\right)\right|}{\left\|\left(p_2^l - p_1^l\right) - \left(p_2^r - p_1^r\right)\right\|} = 2 \times \frac{\left|\left(x_2^l - x_1^l\right) - \left(x_2^r - x_1^r\right)\right|}{\left\|\left(p_2^l - p_1^l\right) - \left(p_2^r - p_1^r\right)\right\|} \tag{5.6}$$

where $||.||$ denotes the vector norm. Note, from its definition δd is always a nonnegative number. Suppose a virtual camera is placed in the middle of the cameras L and R, i.e., at the position of the origin. Since

$$p_1^c = \frac{\left(p_1^l + p_1^r\right)}{2} \text{ and } p_2^c = \frac{\left(p_2^l + p_2^r\right)}{2} \tag{5.7}$$

It follows that:

$$\delta d = \frac{\left|\left(x_2^l - x_1^l\right) - \left(x_2^r - x_1^r\right)\right|}{\left\|\left(p_2^c - p_1^c\right)\right\|} \tag{5.8}$$

Or

$$\delta d = \frac{\left|d_2 - d_1\right|}{\left\|\left(p_2^c - p_1^c\right)\right\|} \tag{5.9}$$

Keep in mind that when the cameras are not coplanar the top portion of equation (5.9) is taken as the magnitude of the distance which is a vector (x and y elements). This term will be incorporated in our optimization and will be explained in the next chapter. The δd value can be used to reveal various stereo-matching constraints. It is thus used as the basis of the unified stereo constraint. A brief summary follows [24]:

- $\delta d > 2$ - violation of order constraint
- $\delta d = 2$ - violation of uniqueness constraint
- $\delta d < 1.1$ or 1.2 - disparity-gradient limit
- $\delta d << 1$ - continuity and figural continuity constraints

5.4 Stereo Matching Techniques

Stereo matching methods can be viewed as a different combination of choices for the following three components:

- A matching token
- A similarity measurement
- A search strategy

The matching token represents the information in the images that will be used for matching (the features that were mentioned before). A basic matching token is the pixel's intensity value; while other tokens can be image features, such as edges and contours. A similarity measurement determines the measure of similarity for each test; different measurements apply to different types of matching tokens, this research focuses on fully incorporating the constraints, plus using an intensity similarity metric that will be discussed later on. A search strategy is how the search area is determined in the target image and how the search is actually carried out. There are two major categories of stereo matching methods based on different matching tokens. The area-based methods and

feature-based methods. In the area-based method, the matching element is template windows of a certain size. The feature-based method aims at establishing the correspondence between a set of image features based on some global optimization as is the case in our work. Correlation functions are used to measure the similarity in area-based methods, while for feature-based methods more complicated criteria are adopted. Search strategies also vary for these two classes of methods, although some principles apply to both. The difference between global optimization search strategies and local search methods is discussed in the next chapter. For now we will briefly mention the differences between area-based algorithms and feature-based ones. Note that even though we match features, we do measure the intensity similarity using correlation based methods which are widely used in area-based algorithms.

5.4.1 Area-based Methods

In area-based methods, the matching is carried out by calculating and comparing correlations between template windows (or some other block matching method). The correlation process is the essential part for an area-based matching algorithm. To match a point p from the left image in the right image: a small window (reference window) is located with p as the center.

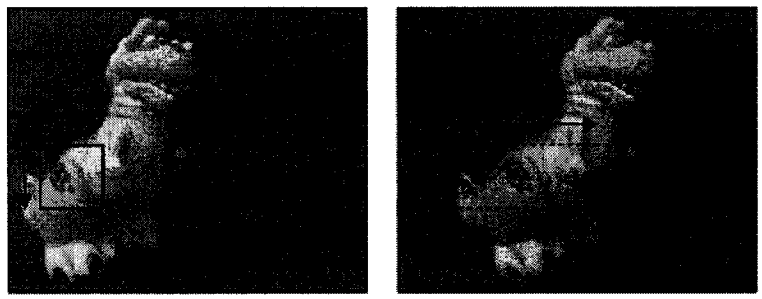

Figure 5.9 Block matching searches one image for the best corresponding region for a template region in the other image. Correspondence metrics are outlined in Table 5-1.

This window is then compared with same sized windows (target window) in the right image for each pixel in the search area. Each comparison produces a correlation score using certain correlation functions. The corresponding pixel p' shall be associated with the window that maximizes the similarity function. This process is shown in Figure 5.9.

Note that there are many methods for comparing template windows; Table 5-1 shows several of these methods. Keep in mind that, as mentioned before, although our method is a feature matching method, we do make use of a template matching "metric" to create a similarity measure in the objective function to be minimized.

Table 5-1. Common Area-based matching methods

MATCH METRIC	DEFINITION
Normalized Cross-Correlation (NCC)	$\frac{\sum_{u,v}\left(I_1(u,v)-\bar{I}_1\right)\cdot\left(I_2(u+d,v)-\bar{I}_2\right)}{\sqrt{\sum_{u,v}\left(I_1(u,v)-\bar{I}_1\right)^2\cdot\left(I_2(u+d,v)-\bar{I}_2\right)^2}}$
Sum of Squared Differences (SSD)	$\sum_{u,v}\left(I_1(u,v)-I_2(u+d,v)\right)^2$
Normalized SSD	$\sum_{u,v}\left(\frac{\left(I_1(u,v)-\bar{I}_1\right)}{\sqrt{\sum_{u,v}\left(I_1(u,v)-\bar{I}_1\right)^2}}-\frac{\left(I_2(u+d,v)-\bar{I}_2\right)}{\sqrt{\sum_{u,v}\left(I_2(u+d,v)-\bar{I}_2\right)^2}}\right)^2$
Sum of Absolute Differences (SAD)	$\sum_{u,v}\left\|I_1(u,v)-I_2(u+d,v)\right\|$
Rank	$\sum_{u,v}\left(I_1'(u,v)-I_2'(u+d,v)\right)$ $I_k'(u,v)=\sum_{m,n}I_k(m,n)<I_k(u,v)$
Census	$\sum_{u,v}HAMMING\left(I_1'(u,v),I_2'(u+d,v)\right)$ $I_k'(u,v)=BITSTRING_{m,n}\left(I_k(m,n)<I_k(u,v)\right)$

In essence, correlation functions give a measure of the degree of similarity between two areas based on the pixels' gray level values. One of the most popular correlation functions is the Normalized Cross Correlation (NCC). NCC calculates the correlation of all the pixels within the range of reference and target window. For a reference window centered on the point to be matched, the target window centered with the matching

point shall produce the highest NCC value among all other windows, which represents the highest similarity. The NCC for each pair of pixels is given by equation in Table 5-1.

5.4.2 Feature-based Methods

A pixel's intensity value is the basic feature of an image. However, a single pixel's gray value does not provide enough information for many applications. More commonly, an image feature refers to a higher level description of an image. The image features are defined as local, meaningful and detectable parts of the image. Local describes the features related to a part of the image with some special properties, and not the global properties of an image. Meaningful means that the features are associated with interesting scene elements via the image formation process. Detectable means that some algorithms exist to detect the feature. The outputs of these algorithms are called the feature descriptors. For example, a descriptor for the line segment feature could consist of the coordinates of the segment's central point, the segment's length and its orientation. The detection of image features can be viewed as a pre-processing stage for matching. In feature-based methods, the matching is based on the numerical and symbolic properties of the features obtained by feature descriptors. Instead of using correlation as similarity measurement, corresponding elements are given by the most similar feature pair using other criteria. It is however possible to use correlation to match those features, in this case the intensity of the neighborhood is used as the similarity metric of the features.

5.4.3 Feature-based versus Area-based Methods

Area-based methods achieve a dense matching result for the stereo image pair, while feature-based methods only match a sparse set of features between the images. However, feature-based methods have the advantage of being efficient and more robust against image variations. The comparison between these two methods is shown in Table 5-2. Note that the aspects that apply to our machine vision application have been highlighted in the table. For instance, the first row denotes the image features where the indoor scenes (assembly line in the case of machine vision applications) are full of straight lines and features that can be easily extracted with a simple edge detector.

Table 5-2 Comparison of area-based and feature-based methods

	Area-Based Methods	Feature-Based Methods
Type of image	Highly textures images, images taken from slightly different viewpoints	Images rich in features such as indoor scenes with many straight lines
Implementation difficulty	Easy	Comparatively more difficult
Pre-processing stage	Not necessary	Feature extraction
Computation time	Calculation of correlation is expensive	Comparatively faster
Sensitivity to noise	Correlation-based methods are sensitive to lighting changes	More robust to illumination changes and other image noise
Matching result	Dense disparity map	Sparse disparity map of matched features

Also notice the fourth row, the gain in the computation time from using feature-based methods outweighs the increased difficulty in the implementation denoted by the second row. The most important is perhaps in the last row, where the results have been explained. We will obtain a sparse depth map in case of a feature-based algorithm, but this is really all that is needed. In machine vision applications as explained before there is no need for full dense maps. In fact, usually around 15-25% of the pixels need to be matched in order to draw enough 3D information from the scene to achieve a given task.

Chapter 6: HNN Position Sensing

6.1 Introduction

Once the cameras have been calibrated and the projection and fundamental matrices are found, in order to proceed with the task of position sensing, we must infer 3D information about the scene. As explained in the previous chapter, this can be done through a stereo matching technique, and once the 3D data is obtained, we can provide motion vectors to a robotics hand [25] in order for it to interact with the environment (i.e. grab various objects in the scene). It was explained that there are different method for stereo matching, feature-based and area-based. It was also shown that most machine vision tasks do not require a full 3D map of the scene, that most machine vision tasks can be accomplished using sparse depth data. Once this data is found, it is only a matter of segmentation of the 3D depth information to obtain information regarding the size and height and location and number of various objects in the scene. In this chapter, we develop a stereo matching technique suitable to our machine vision task using the HNN. However, before proceeding with the algorithm one point has to be cleared. The reader might wonder where the HNN might fit in a stereo matching algorithm. The answer to that is the fact that stereo matching can be viewed as a global search problem that can be solved using a combinatorial optimization method, and HNNs are one such method.

6.1.1 Stereo Matching by Combinatorial Optimization

Stereo matching has traditionally been solved using local search algorithms. Such local search methods could be applied to both feature-based and area-based algorithms. However, recently discreet optimization techniques have become popular in stereo vision and also other fields of computer vision [26]. Such approaches view the stereo matching problem as a global optimization problem where the matching is represented by an energy function which is minimized via a combinatorial optimization technique. But first, let's look at what combinatorial optimization is before proceeding to demonstrate how stereo vision can be formulated as such.

6.1.2 Combinatorial Optimization

Combinatorial optimization problems are concerned with the efficient allocation of limited resources to meet desired objectives when the values of some or all of the variables are restricted to be integral. Constraints on basic resources, such as labor, supplies, or capital restrict the possible alternatives that are considered feasible. Still, in most such problems, there are many possible alternatives to consider and one overall goal determines which of these alternatives is best. For example, most airlines need to determine crews schedules which minimize the total operating cost; automotive manufacturers may want to determine the design of a fleet of cars which will maximize their share of the market; a flexible manufacturing facility needs to schedule the production for a plant without having much advance notice as to what parts will need to be produced that day. The versatility of the combinatorial optimization model stems from the fact that in many practical problems, activities and resources, such as machines, airplanes and people, and pixels in the case of stereo matching, are indivisible. Also, many problems have only a finite number of alternative choices and consequently can appropriately be formulated as combinatorial optimization problems — the word combinatorial referring to the fact that only a finite number of alternative feasible solutions exists. Combinatorial optimization models are often referred to as integer programming models where programming refers to "planning" (not coding) so that these are models used in planning where some or all of the decisions can take on only a finite number of alternative possibilities. Combinatorial optimization is the process of finding one or more best (optimal) solutions in a well defined discrete problem space such as the stereo vision solution space which will be described shortly. Such problems occur in almost all fields of management (e.g., finance, marketing, production, scheduling, inventory control, facility location and layout, data-base management), as well as in many engineering disciplines (e.g., optimal design of waterways or bridges, VLSI-circuitry design and testing, computer vision and etc.). Solving combinatorial optimization problems, that is, finding an optimal solution to such problems can be a difficult task. The difficulty arises from the fact that unlike linear programming, for example, whose feasible region is a convex set, in combinatorial problems, one must search a lattice of feasible points or, in the mixed-integer case, a set of disjoint halflines or line segments to

find an optimal solution. Integer programming problems have many local optima and finding a global optimum to the problem requires one to prove that a particular solution dominates all feasible points by arguments other than the calculus-based derivative approaches of convex programming. This is where HNNs come in and we will show how they can be used to solve this difficult problem.

6.1.3 Local Search versus Global Search in Stereo Correspondence

As explained in the previous section, combinatorial optimization is a problem where we would like to allocate a set of "resources" or "labels" to a set of discrete variables. In the traveling salesman problem [27] which is a classical combinatorial optimization problem, the variables are cities and the labels are the order of which the cities need to be visited. In stereo vision, the variables are pixels from one image and the labels are the pixels from the other image, and we must allocate these labels to our variables in such a way that a global measure of the "goodness" of the solution is minimized. We give this energy function to the HNN in order for it to be minimized effectively (avoiding suboptimal minima).

To get a visual of what are the differences between solving the stereo matching as a traditional search method versus a combinatorial optimization method lets look at Figure 6.1. This figure represents a simplified correspondence problem where we have only three pixels in the left epipolar line and four pixels in the right epipolar line.

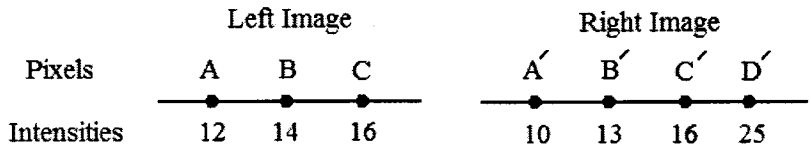

Figure 6.1 Local search versus combinatorial optimization.

We have been given information regarding the gray level of these pixels only (no information regarding their local neighborhood). The task is now to solve this correspondence. Using a local search method the correspondence problem would have been solved as follows:

1. Pick first unmatched pixel
2. Search all candidates on the other image for the best possible match
3. Assign a match and move to step 1

In this case, for illustration purposes we use the direct gray level of a pixel as a measure of similarity (this would be impractical in the real scenario of course). The flaws with using a local search method, is the fact that we do not get the best over all matching. The mathematical expression for a local search in this case would be:

$$\begin{aligned} A \rightarrow B' \quad & \text{Cost} = I(A) - I(B') = 1 \\ B \rightarrow C' \quad & \text{Cost} = I(B) - I(C') = 2 \\ C \rightarrow A' \quad & \text{Cost} = I(C) - I(A') = 6 \end{aligned} \tag{6.1}$$

$$\text{Total Cost} = 9$$

Note that the cost here is defined as the difference in the gray level of the candidate pixel with the original pixel (same metric is used in the global method, with a different search technique). Here the overall cost or the "badness" of the correspondence is calculated as 9, and although the global cost of the whole matching is not optimal, at every single pixel we do pick the highest matching score, thus the name local search. Now let's look at a global matching technique where instead of picking the best choice for a single pixel we pick the best overall arrangement. Note that the combinatorial optimization technique is a method for solving the global matching problem since a greedy search such as in the local method would be extremely inefficient. The local search uses a simple winner-take-all matching strategy which is computationally feasible for a local search method but impractical for a global search strategy since there are too

many combinations. It can all be cleared by looking at the solution to our simple matching problem using a global matching technique:

$$
\begin{aligned}
&A \rightarrow A' \text{ AND } B \rightarrow B' \text{ AND } C \rightarrow C' \text{ Total Cost} = 11 \\
&A \rightarrow B' \text{ AND } B \rightarrow A' \text{ AND } C \rightarrow C' \text{ Total Cost} = 13 \\
&A \rightarrow A' \text{ AND } B \rightarrow B' \text{ AND } C \rightarrow D' \text{ Total Cost} = 9 \\
&A \rightarrow C' \text{ AND } B \rightarrow B' \text{ AND } C \rightarrow C' \text{ Total Cost} = 14 \\
&\text{... (n! combinations)} \\
&\text{Match sequence with lowest cost:} \\
&A \rightarrow A' \quad \text{Cost} = I(A) - I(A') = 2 \\
&B \rightarrow B' \quad \text{Cost} = I(B) - I(B') = 1 \\
&C \rightarrow C' \quad \text{Cost} = I(C) - I(C') = 0 \\
&\text{Total Cost} = 3
\end{aligned}
\tag{6.2}
$$

It is sufficient to compare equations (6.1) with equations (6.2) to see the difference between a local and a global search strategy. We can see that in the global matching methods decisions regarding single matches are not made, rather one single decision is made for all of the correspondences that maximizes the sum of the individual costs of the matching. Obviously if we approach the problem this way we can get the best overall matching where the best decision is made globally (as can be seen in the results of the overall cost or badness of the solution). But the problem here is that there are n! combinations to be considered. Therefore greedy search techniques for finding the optimal solution are impractical. This is where combinatorial optimization techniques are useful. They are to solve such difficult problems in reasonable times, although the solution might not be the best (local minima) but the solution is still close to being the best overall. HNNs as explained in the previous chapters can be used to solve such difficult problems. The details will be presented next. But first, it was mentioned that our algorithm will be a feature-based algorithm[28]. Thus a brief discussion on how feature extraction is performed in this project is warranted.

6.2 Feature Extraction

The stereo algorithm used in this project is a sparse matching algorithm as opposed to a dense matching algorithm where all points in the left image are matched with points in the right image (or vice versa). The reason for this is that, as mentioned earlier, a robotics vision system does not require a full range image (i.e. disparity map) in order to grasp the objects in its field of view. Only a set of critical points on the object need to be resolved in 3D space in order to find an approximate location of the object. This however, requires an extra step in addition to the existing stages of most dense matching algorithm, and that is the feature extraction stage. One of the goals of this research has been to find the optimal method for feature extraction that yields the best results. In other words a set of features need to be found that encapsulate enough information to characterize the object in space and still avoid including too many redundant features. There are three important qualities than any good feature must posses:

- General: represent majority of the useful info in a picture
- Matchable: should be easy to match
- Available: a convenient method for extraction should exist

The feature adopted in this thesis is interest points. There are other features available such as lines and curves, however points are simplest to extract and are sufficient for the task at hand and meet the above three criteria. The approach to finding these points is by applying an interest operator, which selects pixels that are good candidates for matching in the two images. These can be based on gradient, color and so on, and applied quite generally. Several of these features have been tested and the results are shown. The following is a brief discussion of various features that were tested in our experiments.

6.2.1 The Moravec Feature

This operator [29] finds the local maxima of the directional variance minima. The mathematical model for finding such interest points is shown in equation(6.3). The results of this feature extraction can be seen in Figure 6.2.

$$V(i,j) = \min V_\theta(i,j) \quad with\ \theta = \frac{\pi}{4}, \frac{\pi}{2}, \frac{3\pi}{4}, \pi$$
$$V(i,j)_{\frac{\pi}{2}} = \left[f(x,y) - f(x+1,y)\right]^2 \tag{6.3}$$

This figure which will be used later is an image of a book with a cubic object on top of it. Eventually the goal is to find enough information (no more than required) in order to characterize these objects in space (i.e. find their centroid and orientation in 3D space). As you can see the Moravec operator which is actually used in the original work on HNNs and stereo matching [2] finds only very sparse information. Although it is possible to decrease the sensitivity of this feature extractor, its application to machine vision is limited for several reason. In spite of its simplicity this method is very primitive and very sensitive to noise. Nasrabadi and Choo [2] were able to apply this operator because they merely performed their HNN matching on very few points, much less than is required in an actual stereo application such as our position sensing robotics application.

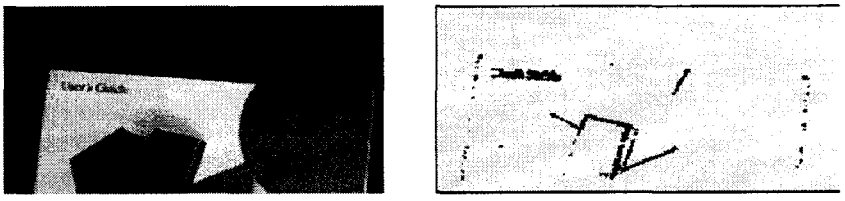

Figure 6.2 A commonly used operator in finding points of interest is the Moravec Operator.

6.2.2 The SUSAN Corner Detector

The SUSAN (Smallest Univalue Segment Assimilating Nucleus) [12] corner detector has become an industry standard for finding corners in digital images. It is simple to implement and its behavior can be find-tuned with a threshold variable. The following shows the quality of the points that have been labeled by the SUSAN corner finder.

Figure 6.3 The SUSAN edge detector only labels the corners (intersection of edges).

It is possible to find finer details with the SUSAN corner detector but still the numbers of features are far from enough for us to be able to completely describe the object in space. This is why several edge detectors were experimented with in order to overcome these shortcomings of the corner detectors. This is one of the ways this work fundamentally differs from the work of Nasrabadi and Choo [2] since there is a real application that needs to be implemented using the stereo matching. That is why simply using a corner detector as they have done is impossible in this case. This makes the matching significantly more challenging than their case since we need to deal with number of features much larger than they have used in their simulations. This is made more difficult since it is known that HNNs perform worse when the dimensionality of the neural network gets larger.

6.2.3 SUSAN Edge Detector

The SUSAN edge detector shares the same principles as the corner detector [12] but it attempts to locate edges rather than corners.

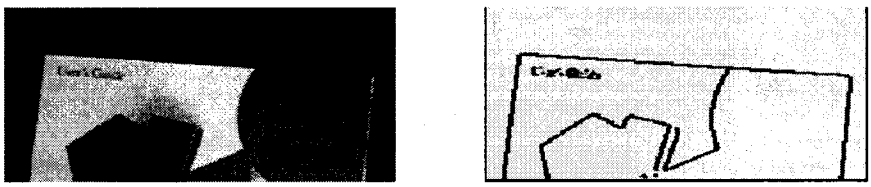

Figure 6.4 The SUSAN edge detector

The experimental results using the SUSAN edge detector showed that it provides enough information regarding the critical points that an object can be characterized effectively; however there was even a better candidate, the Canny edge detector.

6.2.4 Canny Feature Extractor

As an edge detector this is perhaps the most accurate method available. The Canny method [30] differs from the other edge-detection methods in that it uses two different thresholds (to detect strong and weak edges), and includes the weak edges in the output only if they are connected to strong edges. This method is therefore less likely than the others to be fooled by noise, and more likely to detect true weak edges. This method is implemented in the correspondence module, however for every individual image two parameters have to be fixed. This gives us more freedom in determining the amount of information required from the scene. Our scene was particularly difficult for edge extraction since our background was a dusty dark surface. Therefore a noise immune system like the Canny edge detector was particularly useful and was able to provide finer details as shown below.

Figure 6.5 Canny Edge detector

The reader might be wondering why we have not discussed the Sobel edge detector. This particular edge detector is perhaps the most widely used edge detector, and perhaps one of the most unsuitable one to this application. As explained our background is a dusty dark surface. The results of the Sobel edge extraction were quit poor as can be seen below.

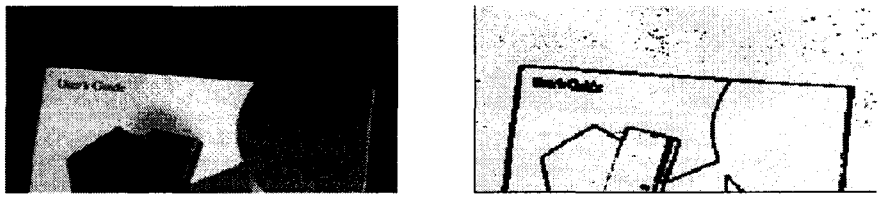

Figure 6.6 Sobel edge detector, poor noise immunity

One might argue that the added improvement in performance in the case of the Canny edge extractor comes at the price of higher computational time. But this is misleading since the added improvement in performance is worth the added computation time (which is in fact very minimal). Note that in cases when the Sobel extractor was used the application performed incorrectly since the depth of noise points offset the location of the centroids that have been found and the robot would be provided with incorrect locations. This is why it is essential to use the best performing feature extractor. Also note that the more features to match, the more ambiguity is present in the system. This forces us to keep the number of feature as low as possible, yet high enough to provide the required information.

6.2.5 Matching Strategy

As mentioned in the introduction, there has been a surge of stereo matching algorithms that, unlike the correlation based methods, view the matching process as a combinatorial optimization problem [5, 26]. This is inspired by the fact that stereo matching can be viewed as a labeling where every primitive on the left image is assigned one from the right image such that a cost function is minimized. Formulating the stereo algorithm as such will have the benefit of providing an optimal mapping of primitives

from one image to another without being affected by local ambiguity or lack of texture. The most important part of such an energy minimization would be the cost function. This cost function is devised as to make use of stereo constrains in such a way that the matching will be in compliance with the stereoscopic image properties. In the following sections we will discuss our novel objective function which uses Disparity Gradient (DG) and intensity similarity as its main constraint and forces matches that only follow the epipolar geometry.

6.2.6 Objective Function

Nasrabadi and Choo [2] devised a HNN formulation of the stereo matching algorithm; however their algorithm does not make explicit use of the epipolar geometry, Disparity Gradient or intensity similarity. In this work we have enhanced their objective function by adding three important cost terms, and also by using a continuous HNN rather than a discrete one. This is due to the fact that a continuous HNN is more likely to achieve a global minim without getting stuck in local minima. It is well known that the Lyaponov energy function of a two dimensional HNN is defined as [2]:

$$E = \left(-\frac{1}{2}\right)\sum_{i=1}^{N_l}\sum_{k=1}^{N_r}\sum_{j=1}^{N_l}\sum_{l=1}^{N_r} T_{ikjl} V_{ik} V_{jl} - \sum_{i=1}^{N_l}\sum_{k=1}^{N_r} I_{ik} V_{ik}. \tag{6.4}$$

Now the problem becomes that of mapping the stereo constraints to the Hopfield Energy function in order to minimize it and achieve correspondence. Note that the HNN is a set of fully interconnected neurons whose activations are denoted by V_{ik} where "i" denotes the row number in the 2D neural grid and "j" denotes the column number in the grid. After the energy function has been properly mapped the network will evolve and the final state of the network will provide the correspondence.

The correspondence is established by noting the value of every neuron. For instance, if the activation of neuron V_{34} is found to be the highest in its row and column, then it is deduced that the 3rd edge point from the left (or right) is matched with the 4th edge from the right (or left) image as shown in Figure 6.7. This is similar to how other combinatorial optimization problems are solved using the HNN [27].

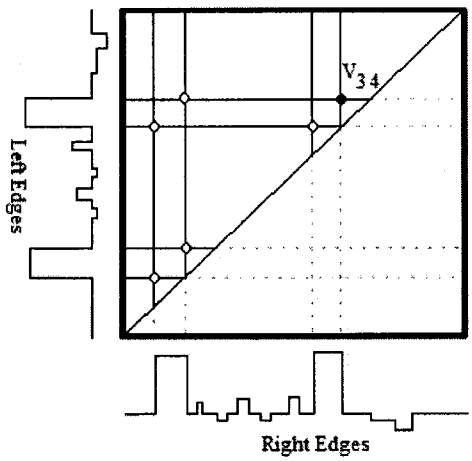

Figure 6.7 Matching by HNN

The following is the energy function used in this work which combines two additional constraints to the one used in [2]:

$$E = -A\sum_{i=1}^{N_l}\sum_{k=1}^{N_r}\sum_{j=1}^{N_l}\sum_{l=1}^{N_r} C_{ikjl}V_{ik}V_{jl} + B\sum_{i=1}^{N_l}\left(1-\sum_{k=1}^{N_r}V_{ik}\right)^2 + C\sum_{k=1}^{N_r}\left(1-\sum_{i=1}^{N_l}V_{ik}\right)^2 + D\sum_{i=1}^{N_l}\sum_{k=1}^{N_r}\tau_{ik}V_{ik}. \quad (6.5)$$

At this point equations (6.5) and (6.4) can be thought of as separate entities. Later it will be shown that by rearranging (6.5) we can use the form of (6.4) and thereby use the HNN as our energy minimization tool. The first term contains information regarding the compatibility of two matches, that of V_{ik} and V_{jl}. These matches are related by their connection value which contains information regarding the intensity similarity and Disparity Gradient. Here we define the connection of the neurons as:

$$T_{ikjl} = \frac{2}{(1+e^{\lambda(X-\theta)})} - 1 \tag{6.6}$$

This value is a measure of compatibility based on the variable X which encodes information about the match. In this work X is defined to be:

$$X = \frac{1}{W_1 g_{ikjl} + W_2(M_{ik} + M_{kl})} \tag{6.7}$$

Here the g_{ikjl} term denotes the Disparity Gradient compatibility between the two matches "ik" and "jl". This value is a function of the actual Disparity Gradient taken from Li and Hu [24] to be:

$$g_{ikjl} = 2e^{-\delta d/T} - 1 \tag{6.8}$$

Where δd denotes Disparity Gradient and T is a constant that is chosen as in [24] to be 1.5. For a complete discussion of Disparity Gradient and its use of stereo matching see Li and Hu [24] and Pollard [22]. Here we have added another term that includes information regarding the intensity similarity. This is used in order to reduce the ambiguity of matches since Disparity Gradient will not always provide enough information for choosing the right match and will lead to some false matches. Each intensity similarity term is obtained using normalized cross correlation as follows:

$$M_{ik} = \frac{\sum_{u,v}[f(x_l+u,y_l+v)-\bar{f}][t(x_r+u,y_r+v)-\bar{t}]}{\sum_{u,v}[f(x_l+u,y_l+v)-\bar{f}]^2 \sum_{u,v}[t(x_r+u,y_r+v)-\bar{t}]^2} \tag{6.9}$$

Where u and v define a neighborhood area around the points x_l and x_r which is the point belonging to the ith feature of the left image and the points x_r and y_r which are the location of the kth feature of the right image. Since only edge points will be matched we can use a window size of 7 or 5 without loosing a significant amount of accuracy. This is due to the fact that edge feature can be much more reliable than points with neighborhoods of low variance. Also it must be noted that W_1 and W_2 are weights that

can be chosen experimentally. The best combination in our experiments were W_1=0.3 and W_2=0.7.

The second and third terms tend to enforce the uniqueness constraint since the probabilities (states of neurons) in each row or column should add up to 1. We know the probability represents a measure of match between a feature point in the left image and that of the right image. Therefore only one neuron in each row or column should be set to "on" according the uniqueness constraint. This is the reason for the second and third term to prevent the probability of multiple matches. This uniqueness problem is one of the reasons we chose a continuous HNN instead of a discrete one as in [2]. A discrete HNN will evolve to a state where the activations will either be set to one or zero. Thus it is likely to obtain multiple "on" activations in one row or column, as is the case in [2]. However using a continuous scheme we avoid this pitfall by choosing the highest value in a row since activations can take on a continuous value in the range of zero to one.

The fourth term of the objective function tends to allow matches that are within an acceptable limit to the epipolar line. Although mostly new stereo algorithms assume the images have been calibrated accurately and that images have been precisely rectified, this can not be assumed in every scenario especially in machine vision application. This is because we wish to have the freedom of non-coplanar cameras while at the same time we can not guarantee accurate calibration. Therefore we can only assume weak calibration and allow certain variability for a candidate match in regard with the found epipolar line. This will give us the freedom of not having an absolutely correct fundamental matrix. The term τ is basically the perpendicular distance of a candidate match to the obtained epipolar line (not the real epipolar line):

$$\tau_{ik} = \frac{|ax_r + by_r + c|}{\sqrt{a^2 + b^2}} \tag{6.14}$$

Where x_r and y_r are the location of the kth feature point in the right image and the parameters a, b and c are those of the epipolar line in the right image corresponding to the ith feature point in the left image.

6.2.7 HNN Parameters

Once the objective function has been defined all that is left is to construct a HNN of the appropriate size, set the weights and let the network evolve. An HNN that would require matching for N_l and N_r would require $N_l \times N_r$ nodes and $(N_l \times N_r)^2$ connection weights. However the there is no self connection (i.e. $T_{ikik}=0$) and the weights are symmetric (i.e. $T_{ikjl}= T_{jlik}$). So although some memory usage can be spared due to these facts, the HNN remains a very memory intensive algorithm. This could be one of the reasons why there still has not been a single dense disparity matching algorithm using the HNN.

Once an HNN of the right size has been set up the weights will be set according to the objective function. However, our objective function needs to be rearranged in order to achieve the form of (6.4) so we can use it to determine connection weights and biases. After some algebraic manipulation we arrive at this form of (6.4):

$$E = \left(-\frac{1}{2}\right)\sum_{i=1}^{N_l}\sum_{k=1}^{N_r}\sum_{j=1}^{N_l}\sum_{l=1}^{N_r} W_{ikjl} V_{ik} V_{jl} - \sum_{i=1}^{N_l}\sum_{k=1}^{N_r} I_{ik} V_{ik} \tag{6.10}$$

Where the weights are:

$$W_{ikjl} = 2(AC_{ikjl} - B\delta_{ik} - C\delta_{jl}) \tag{6.11}$$

And the biases are:

$$I_{ik} = B + C - D\tau_{ik} \tag{6.12}$$

Where A, B, C and D are free parameters of the system. There is no mathematical way of finding their optimum value. In this work we mostly use small positive values.

The best results that were found in this work were 2, 1, 1 and 1 were used respectively for A, B, C and D.

6.2.8 HNN Evolution

Note that our updating differs from Nasrabadi and Choo's method [2] since they used an updating scheme where a neuron is updating in a window of a certain size using other neurons at a predetermined distance since they realized a neuron related to a match from one part of an image should not be affected by one from a point that is unlikely to belong to the same object. Using this scheme they avoid interference between unrelated matches thus they resort to an unorthodox updating scheme which does not necessarily achieve a global optimum. However in our scheme since we added a term for epipolar distance it is unlikely that unrelated matches affect one another. Another method that we attempted was to scale down weights connecting far away points however the results were not satisfactory perhaps due to not being able to fine tune the network parameters. Figure 6.8 shows the evolution of a continuous HNN. It is proven by the Lyaponov function of the HNN that the network's energy is guaranteed to decreased, thus, its usage as an energy minimization tool.

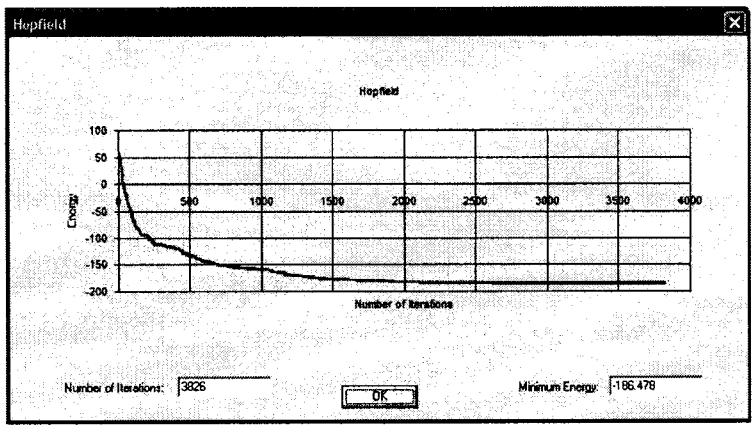

Figure 6.8 Energy of an HNN

Note the same energy function solved using a binary HNN shown in Figure 6.9. It is always possible to achieve a lower energy level using a continuous HNN instead of a discrete HNN as used by Nasrabadi and Choo [2].

Once the HNN has evolved we can let the network evolve for a few hundred iterations to ensure the HNN is in a stable point. Once this stable point has been reached we can infer information about the correspondences in the stereo image pairs using the state of the outputs of the HNN. The next section describes the results and how this information is used to provide motion vectors to the robot.

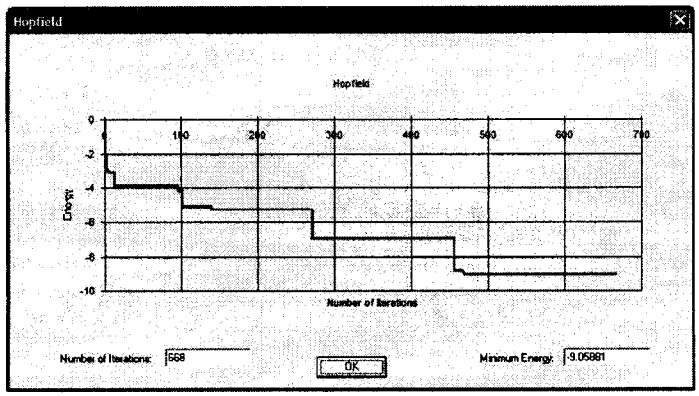

Figure 6.9 Energy of a discrete HNN

The state of our 2D HNN can be inferred by moving horizontally and vertically and picking the highest values in each row and column (ensuring that every row or column has one value tagged in order to adhere to the uniqueness constraint) as shown in Figure 6.10. This table shows the values after they have been pruned and the winning values have been converted to 1s and others to 0s. Keep in mind that in reality all cells are filled with continues values. Note that there are unmatched pixels, this means the output of no neuron was high enough to denote a match (after thresholding the values).

0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0

Figure 6.10 Output of an HNN after settling in a stable state

6.2.9 Position Sensing

Once the correspondences have been established between the features we can find their 3D locations in space. Our HNN can provide us with this information. It is important for the readers to realize that this result is not always correct. This might come as a surprise but in addition to the fact that stereo matching is an ill posed problem; the HNNs have traditionally been lagging in their application in combinatorial optimization problems [27, 31]. Although it is true that there has been a great amount of literature focused purely on trying to improve the performance of the HNN, but still if accuracy is the primary goal of an actual application perhaps dynamic programming or graph cuts [5] algorithm could be used instead of the HNNs.

The primary reason for our exploration of the HNNs in this machine vision application has been the fact that HNNs can be easily realized in VLSI analog chips and provides extremely rapid solution rates. This is to say that HNNs can provide their advantage when implemented in hardware since they are very promising for real-time application. However when implementing in software they use a great amount of memory since the network is fully connected and the number of nodes can become exceedingly large. And then there is the problem of infeasible solutions [27]:

"...[despite recent improvements in HNN], the reputation of the Hopfield network for solving combinatorial optimization problems does not appear to have been

resurrected. Recent results have shown that, unless the TSP is Euclidean, the quality of the solutions found using a Hopfield network is unlikely to be comparable to those obtained using traditional techniques".

And also [31]:

"...we show that network dynamics are often ill suited to the solution of other problems [other than TSP]. In addition, the use of alternative objective functions does not reliably improve performance. It appears that neural networks are not well adapted to the solution of [combinatorial optimization] problems without an underlying geometric structure; they are therefore not as attractive as they might have originally seemed."

Therefore when implementing an HNN algorithm for solving the stereo problem the reader should be careful about the infeasible solutions that the system might provide. One of the reasons for our system's functionality in spite of the downside of using HNNs has been the fact that there is a large amount of redundancy in the system. We use cubic objects with flat surfaces to test our experiments, thus it is relatively easy to prune out outliers using a simple clustering method that is outlined later on. For now let's look at an example to demonstrate the results of the algorithm. Figure 6.11 shows several blocks that were used in a sample run of our vision application. The calibration has already taken place before proceeding with the actual position sensing.

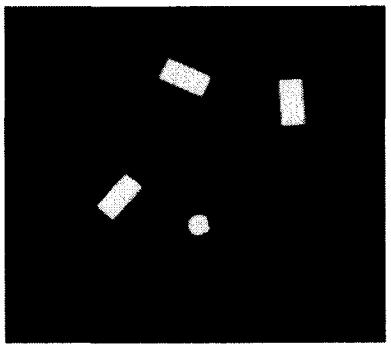

Figure 6.11 Metallic blocks used to test the machine vision application

The first step would be to use the feature extractor to extract the edges of the objects in the scene as shown in Figure 6.12. Following this the HNNs weights will be

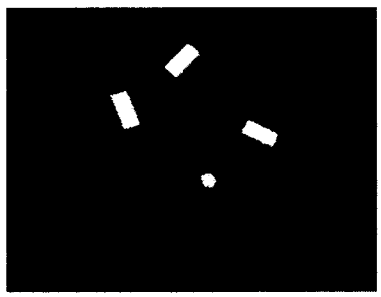

Figure 6.12 Sample blocks and features

set according to equations (6.11) and (6.12). Once the 3D coordinates have been found using the stereo matching paradigm we can infer information regarding the number of objects in space and their orientation and also their centroids. This information is enough for the robot to grab these objects. This segmentation of the depth data is done using a clustering algorithm that operates on the depth data as will be explained in the next chapter.

6.2.10 Clustering the 3D Points

Once 3D data are available regarding all feature points this data has to be processed. The input of this stage (the clustering) is the raw 3D data and the output is the number of objects plus their centroid and orientation. Note that we have assumed simple cubical objects of different heights, which means this process would only apply to such shape and more complicated algorithm need to be developed for objects of arbitrary shape. The basic pseudo-code of the algorithm which is based on K-mean clustering is as follows:

- initialize the clusters:
  - num_objects=0
  - for all j → Cluster[j].Z=0
- for any 3D point P[i] :
- if p[i].Z belongs to cluster k (within a threshold from its mean)

- add point p[i] to cluster "k"
- else
  - start a new cluster : num_objects ++
  - add the point p[i] to cluster "num_objects"

6.2.11 Orientation in Space

Once this process has been completed and every point has been given to a cluster, if two clusters have average depths within a threshold, we merge them since the height difference between two objects will be within a threshold. This is threshold can be found for a given scene using trial and error. Also if a cluster has less than a threshold number of points, it will be removed since is it too small to be considered objects (most likely noise). These clusters now represent the actual objects' centroids. Following this the variable "num_objects" contains the number of objects in space. Therefore every pixel has now been labeled an objects number and now it is known which objects each pixel belongs to. This information can now be used to find the orientation of these objects. This is done using the concept of moments with the equation:

$$\theta = \frac{1}{2}\tan^{-1}\frac{2\mu_{11}}{\mu_{21}-\mu_{02}} \tag{6.13}$$

Where μ_{ij} is the ijth central moment of all the 3D points in a given cluster. Also this angle is the angle of the objects' orientation in the xy plane. That is the surface of the objects is assumed to be parallel to the xy plane. After the centroid and the orientation has been found for every single object some sanity checks can be performed on the data before providing the motion vectors to the robot. For instance we can check for the average depth of every candidate object, we know it has to be within a limit. For instance we know the working environment is a certain size and that for instance no objects can be bigger than that size (in our case 20 cm). Also we must ensure no object with the depth of zero (ground level) is accepted. Such artifacts occur frequently and are results of the features detector picking pixels from the noisy background. This could be avoided if a background subtraction was performed; however discarding zero-depth data is more straightforward.

6.3 Experiments

In our experiments as mentioned before several cubic objects of varying heights were used in every run of the program (as shown in Figure 6.11). After feature extraction and matching using the HNN, the clustering algorithm is performed on the depth data. Following this the centroids of the objects and their orientation has been found and outliers have been removed using the clustering algorithm. The results are shown on the screen to the user for debugging purposes but the real output of the program is the motion vectors provided to the robot. However, having obtained the centroids of the objects and their orientation, moving the robot to the actual location is a matter of technology (the type of robot, the working environment ...) and will not be discussed here. The interested reader can consult the thesis by Cardillo [25] to obtain information regarding how the robotics commands were calculated.

Following is an example of the results shown to the user; Figure 6.13 shows the features of the objects whereas Figure 6.14 shows the features that have actually been matched (note that the matching processes fails to match a large number of points). This does not yield poorer results since the unmatched points are evenly distributed throughout the image and do not (usually) offset the final results. Figure 6.15 shows the orientation (where the straight line passes through the centroid). The attentive reader might wonder how the straight line denoting the orientation has been drawn, since the orientation is an angle in the world xy plane rather than the "uv" image plane. This is done by finding the equation of the line in 3D space, and then projecting it into the image plane using the projection matrix.

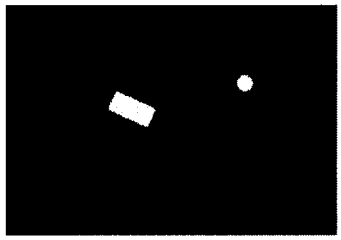

Figure 6.13 Features of the image (red pixels have been artificially added to show the feature locations).

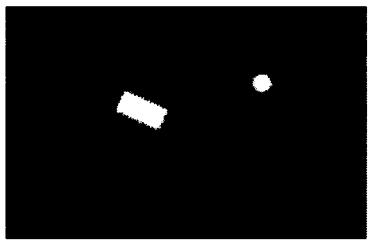

Figure 6.14 Features that have been matched (much less red pixels).

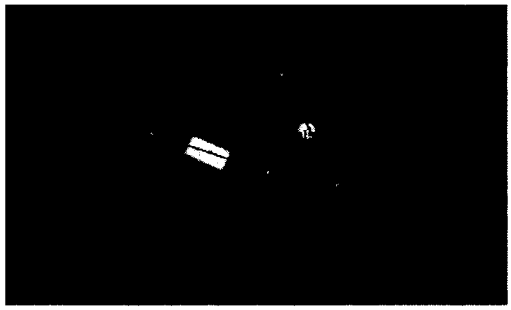

Figure 6.15 Centroid and orientation of the object.

Also it is important to see the depth histogram to get a feeling for the functionality of the clustering algorithm. The depth histogram for the previous example (Figure 6.15) is shown below; note the high amount of noise (mismatched points).

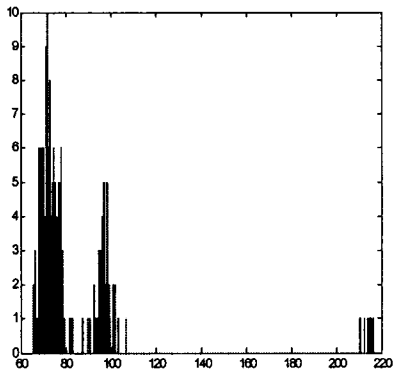

Figure 6.16 Depth histogram of the objects in Figure 6.15

In an ideal situation there should only be two spikes in the histogram, but because of various errors (calibration error, epipolar localization error, matching error...) the shapes are somewhat smeared and there is also an artifact (the object to the far right). We have two safety measures against such artifacts; one is the fact that the numbers of pixels belonging to it are less than our threshold as mentioned before. And another which applies to this example is that the mean height of the artifact object is larger than our maximum allowable heights. In order to better see the purpose of the clustering stage note another example where artifacts are more significant. Consider the objects in Figure 6.11. The raw data pertaining to the objects has been drawn using a 3D plotting utility in MATLAB as shown in Figure 6.17.

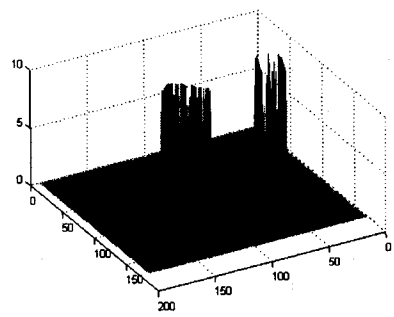

Figure 6.17 Raw depth points.

You can see that many points have either not been matched (note the sparse set of data along object boundaries) or have been matched incorrectly. The challenge is to reduce the destructive effects of the outliers and also to use the remaining correct data to interpolate the depth of the unmatched pairs and to build a model of the object in order to manipulate its location use a robotics arm. We can see how the clustering algorithm performs. The first figure shown below is an example of applying the clustering algorithm with incorrect parameters.

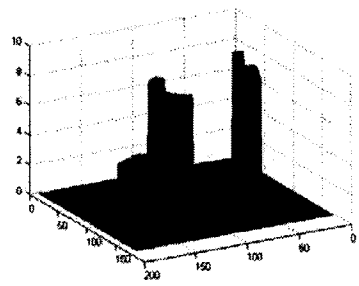

Figure 6.18 Clustering with suboptimal parameters

There are two essential parameters to our clustering algorithm. First the number of pixel threshold that would constitute a real object and also the threshold of belonging to a cluster (how close does a particular data have to be to the mean of a particular cluster in

order for it to belong to that cluster). These parameters can be simply found using trial and error. Now lets take a second look at performing the same operation this time using a better set of parameters:

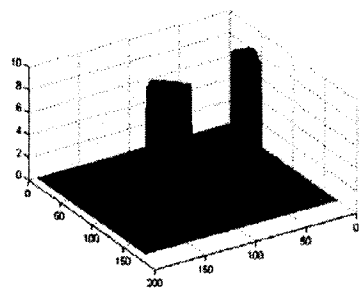

Figure 6.19 Clustering using correct parameters

Now we have built correct models of data and have been able to distinguish between their heights and also remove the outliers. All that is left to do is to convert this information to robot motion vectors. One more test of the clustering algorithm can be performed in order to show the validity of the system. This is done by using an arbitrarily size object. This time Gaussian noise was added to the fundamental matrix in order to purposefully confuse the matching process. The following figure shows the object which has an irregular shape.

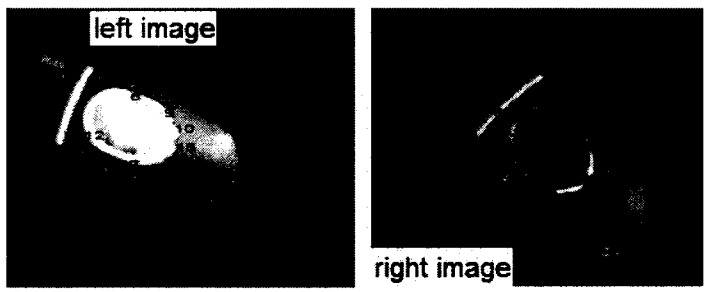

Figure 6.20 Arbitrary shaped object used for testing.

Below is the result of the clustering of the data. It can be seen that there is significant amount of noise and mismatched points. However the system was able to cerate a rough model of the object, which is in fact accurate enough for our machine vision task. Note that the algorithm was able to perform reasonably well in the absence of accurate epipolar geometry, this is due to the fact that the epipolar geometry is incorporated as a soft constraint in our algorithm (the closer the candidate to the epipolar line the higher the score as shown in equation (6.14)) and that we have avoided using the epipolar line as a hard constraint.

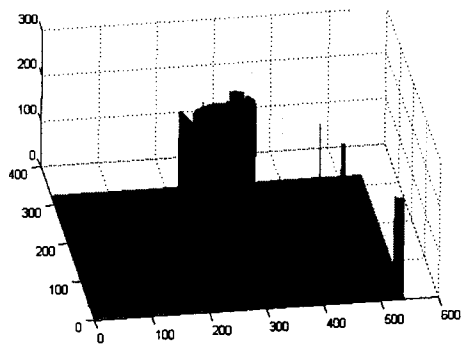

Figure 6.21 Result of clustering the depth data in case of an arbitrary shaped object

In spite of this, the algorithm does begin to fail when the number of objects is too high or when the height difference between the different objects is not large enough.

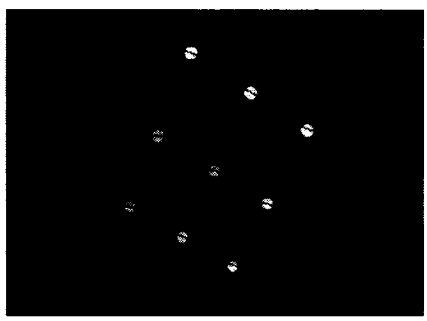

Figure 6.22 System is capable of processing higher number of objects

Figure 6.22 is another example of the clustering algorithm performed for an object with nine different marks on it (effectively there are nine different objects). In this case the parameters of the clustering algorithm had to be adjusted in order to accommodate the new problem. The figure also shows the orientation and centroid lines for the found objects. The depth data that has been found is still accurate enough that we are able to distinguish between the objects and make decisions regarding their centroids and orientation. However in this case since the objects are circular the orientation is a meaningless value and the orientation lines merely reflect noise. To better see the accuracy of the data note the depth histogram of these objects shown below.

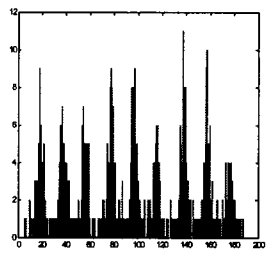

Figure 6.23 Depth histogram of nine objects

Using this histogram and our clustering algorithm we are still able to build a 3D model of the objects in the scene and use the robotics system to grab them. The following shows the 3D model built using MATLAB.

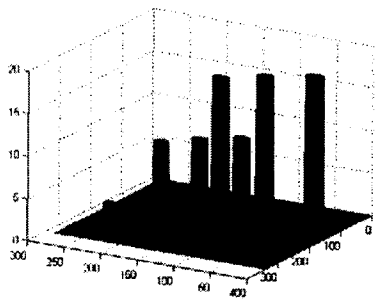

Figure 6.24 3D model of nine objects

Another way that our implementation allows us to test the output of the program is accomplished by a selective matching process. This is done by selecting single points, having the program match them using a simple local search using some type of template matching. Below is an example of how this works in our implementation, note that this is mainly useful for testing the calibration data and checking heights of objects one by one. Once the system has found the match various statistics regarding the match and other data in the system are printed on the screen as shown in Figure 6.25.

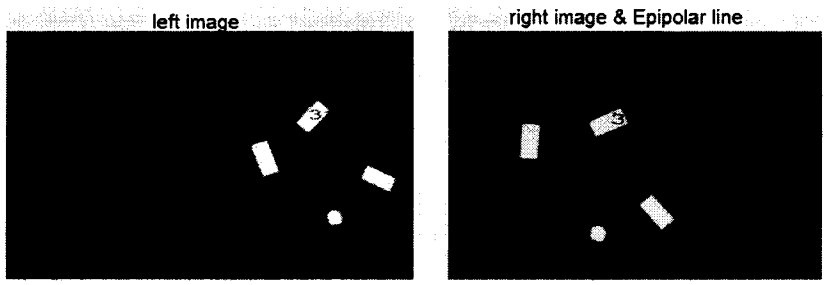

Figure 6.25 Selective point matching to check the performance of the system.

Chapter 7: Conclusion

The need for accurate and fast machine vision algorithms are ever increasing. One of the most challenging problems of 3D Vision is the problem of correspondence where points are matched from one viewpoint to another in order to infer 3D information regarding the scene. Many algorithms have been devised for performing this task. These algorithms suffer from either of the two flaws. The first type of algorithms view the matching as a local search and attempts to increase the accuracy by incorporating as many constraints as possible (traditional methods). These algorithms suffer from lack of accuracy since they use local search rather than global search which always performs better. The second set of algorithms view the matching as a global search and attempt to solve it by using a combinatorial optimization method. Several such algorithms have been devised which perform extremely well [5]. However, their flaws lay in the fact that most combinatorial optimization methods are computationally expensive and complicated (in comparison with local search methods). This makes them unsuitable for hardware implementation as well as for applications in real-time machine vision.

Our attempt in this thesis has been to built upon previous work [2] and take the HNNs one step further as tools for stereo matching. Our attempt is to use an HNN as a combinatorial optimization too in spite of its numerous shortcomings [27, 31]. The reason for our exploration of this tool has been the fact that HNNs have comparatively simple analog VLSI implementations and if they are proven to perform anywhere near other combinatorial optimization algorithms for stereo matching it will lead to a new generation of global matching tools that can easily be implemented in hardware and provide extremely high solutions rates.

We have built upon the work of Nasrabadi and Choo mainly by modifying their objective function and also by incorporating the HNN in a real world machine vision application which puts significant constraints in the performance of the algorithm. We have applied their algorithm to non-coplanar camera geometry and have incorporated the

epipolar constraint as a soft constraint as well as using the Disparity Gradient constraint in our objective function. We have also avoided the use of binary HNN since they are known to be inferior to continuous HNNs.

7.1 Future Work

The fact that HNNs are ever evolving makes them very attractive subjects for research. However the researcher must be aware of the facts that many forms of HNNs perform worse in most cases in comparison with other combinatorial optimization techniques. There has recently been a number of works [27, 32, 33] who claim to have improved the performance of the HNNs to the point where they can perform as well as other optimization techniques.

Experimentation with newer forms of HNNs in stereo vision and also modification of the HNN itself (as many have attempted) could prove to be an interesting research topic.

REFERENCES

1. Chia-Horng, H. and W. Jung-Hua. *Stereo correspondence using Hopfield network with multiple constraints.* in *2000 IEEE International Conference on Systems, Man, and Cybernetics.* 2000. Nashville, TN.
2. Nasrabadi, N.M. and C.Y. Choo, *Hopfield network for stereo vision correspondence.* IEEE Transactions on Neural Networks, 1992. **3**(1): p. 5-13.
3. Ruichek, Y., *Multilevel- and neural-network-based stereo-matching method for real-time obstacle detection using linear cameras.* IEEE Transactions on Intelligent Transportation Systems, 2005. **6**(1): p. 54-62.
4. Haykin, S.S., *Neural networks: a comprehensive foundation.* 1994, New York: Macmillan.
5. Scharstein, D., R. Szeliski, and R. Zabih, *A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms.* International Journal of Computer Vision, 2002. **47**(1-3): p. 131-140.
6. Sonka, M., V. Hlavac, and R. Boyle, *Image processing, analysis, and machine vision.* 2nd ed. 1999, Pacific Grove, CA: PWS Pub.
7. Birchfield, S. *An Introduction to Projective Geometry.* http://vision.stanford.edu/~birch/projective/ (Jan, 2006)
8. Hartley, R. and A. Zisserman, *Multiple view geometry in computer vision.* 2nd ed. 2003, Cambridge: Cambridge University Press.
9. Forsyth, D. and J. Ponce, *Computer vision: a modern approach.* Prentice Hall series in artificial intelligence. 2003, Upper Saddle River, N.J.: Prentice Hall.
10. Tsai, R., *A versatile camera calibration technique for high-accuracy 3D machine vision metrology using off-the-shelf TV cameras and lenses.* IEEE Journal of Robotics and Automation, 1987. **3**(4): p. 323-344.
11. Press, W.H., *Numerical recipes in C: the art of scientific computing.* 1992, Cambridge; New York: Cambridge University Press.
12. Brady., S.M.S.a.J.M., *SUSAN - a new approach to low level image processing.* International Journal of Computer Vision, 1997. **23**: p. 45-78.
13. Otsu, N., *A threshold selection method from gray level histograms.* IEEE Transactions on Systems, Man and Cybernetics, 1979. **9**: p. 62-66.
14. Zhang, Z., *A flexible new technique for camera calibration.* IEEE Transactions on Pattern Analysis and Machine Intelligence, 2000. **22**(11): p. 1330-1334.
15. Intel Corporation, *http://sourceforge.net/projects/opencvlibrary/* (Jan, 2006).
16. Kolmogorov, V. and R. Zabih, *Computing visual correspondence with occlusions using graph cuts.* International Conference on Computer Vision, 2001. **2**: p. 508-515.
17. Kolmogorov, V. and R. Zabin, *What energy functions can be minimized via graph cuts?* IEEE Transactions on Pattern Analysis and Machine Intelligence, 2004. **26**(2): p. 147-159.
18. Kohonen, T., *Self-Organizing Maps.* Second ed. Series in Information Sciences. Vol. 30. 1997: Springer, Heidelberg.

19. Hopfield, J.J., *Artificial neural networks.* IEEE Circuits and Devices Magazine, 1988. **4**(5): p. 3-10.
20. Hopfield, J.J. and D.W. Tank, *Neural Computations of Decisions in Optimization Problems.* Biological Cybernetics, 1985. **52**: p. 141-152.
21. Prâincipe, J.C., N.R. Euliano, and W.C. Lefebvre, *Neural and adaptive systems: fundamentals through simulations.* 2000, New York: Wiley.
22. Pollard SB, M.J., Frisby JP., *PMF: a stereo correspondence algorithm using a disparity gradient limit.* Perception, 1985. **14 (4)**: p. 449-470.
23. Burt, P. and B. Julesz, *A disparity gradient limit for binocular fusion.* Science, 1980. **208**(4444): p. 615-617.
24. Li, Z. and G. Hu, *Analysis of disparity gradient based cooperative stereo.* IEEE Transactions on Image Processing, 1996. **5**(11): p. 1493-1506.
25. Cardillo, J., *3-D robot vision metrology and camera calibration from focus information*, in *Electrical and Computer Engineering.* 1988, University of Windsor: Windsor.
26. Brown, M.Z., D. Burschka, and G.D. Hager, *Advances in computational stereo.* IEEE Transactions on Pattern Analysis and Machine Intelligence, 2003. **25**(8): p. 993-1008.
27. Smith, K., M. Palaniswami, and M. Krishnamoorthy, *Neural techniques for combinatorial optimization with applications.* IEEE Transactions on Neural Networks, 1998. **9**(6): p. 1301-1318.
28. Trucco, E. and A. Verri, *Introductory techniques for 3-D computer vision.* 1998, Upper Saddle River, NJ: Prentice Hall.
29. Moravec., H.P., *Towards Automatic Visual Obstacle Avoidance.* Proc. 5th International Joint Conference on Artificial Intelligence, 1977: p. 584-585.
30. Canny, J., *A Computational Approach to Edge Detection.* IEEE Transactions on Pattern Analysis and Machine Intelligence, 1986. **8**: p. 679-698.
31. Gee, A.H. and R.W. Prager, *Limitations of neural networks for solving traveling salesman problems.* IEEE Transactions on Neural Networks, 1995. **6**(1): p. 280-282.
32. Huajin, T., K.C. Tan, and Y. Zhang, *A columnar competitive model for solving combinatorial optimization problems.* IEEE Transactions on Neural Networks, 2004. **15**(6): p. 1568-1574.
33. Galan-Marin, G. and J. Munoz-Perez, *Design and analysis of maximum Hopfield networks.* IEEE Transactions on Neural Networks, 2001. **12**(2): p. 329-339.

APPENDIX A: SOURCE CODE

```
// camera.cpp: implementation of the camera class.
//a class for performing various camera related tasks
//
//////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "calibrb.h"
#include "camera.h"
#include <math.h>
#include <fstream>
#include <string>
#include <iostream>
#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif

inline double FMIN(double a, double b) { return a<b?a:b; }
inline double FMAX(double a, double b) { return a>b?a:b; }
inline int IMIN(int a, int b) { return a<b?a:b; }
inline int IMAX(int a, int b) { return a>b?a:b; }


#define TOL 1.0e-28
#define BORDER_MAX 100000
#define CIRCLES_MAX 10
using namespace std;
//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

static double at,bt,ct;
#define PYTHAG(a,b) ((at=fabs(a)) > (bt=fabs(b)) ? \
(ct=bt/at,at*sqrt(1.0+ct*ct)) : (bt ? (ct=at/bt,bt*sqrt(1.0+ct*ct)): 0.0))

static double maxarg1,maxarg2;
#define MAX(a,b) (maxarg1=(a),maxarg2=(b),(maxarg1) > (maxarg2) ?\
  (maxarg1) : (maxarg2))
#define SIGN(a,b) ((b) >= 0.0 ? fabs(a) : -fabs(a))
double *vector(int n){

 double* vec=new double[n+2];

  return(vec);
```

```
}
double **matrix(int m, int n){

  /* create a matrix, for example matrix (1, 3, 1, 4); */

 int i;
 double **mat;
 mat= new double* [m+2];


 for(i=0; i<m+2; i++)
  {
  mat[i] = new double[n+2];
  }


 return(mat);
}
void freevector(double* vec)
{

delete [] vec;
}

void freematrix(double** mat,int num_rows){
 int i;

 for(i=0; i<num_rows+2; i++)
  {
  delete [] mat[i];
  }


delete []  mat;

}

camera::camera()
{
 int z,xx,yy;
 for(z=1;z<=8;z++){
   switch(z){
   case 1:
     xx=-1;
     yy=-1;
     break;
   case 2:
     xx=0;
     yy=-1;
     break;
   case 3:
     xx=1;
```

```
      yy=-1;
      break;
    case 4:
      xx=1;
      yy=0;
      break;
    case 5:
      xx=1;
      yy=1;
      break;
    case 6:
      xx=0;
      yy=1;
      break;
    case 7:
      xx=-1;
      yy=1;
      break;
    case 8:
      xx=-1;
      yy=0;
      break;
  }

  neighbour1[z-1].x=xx;
  neighbour1[z-1].y=yy;
  }
}

camera::~camera()
{

}
HBITMAP camera::ArrayToBitmap(unsigned char *p)
{

  int width1=S_X;
  int height1=S_Y;


  void* m_pBits;
  HDC m_hdc ;
  HBITMAP  hBitmap;
  int i;
BITMAPINFO *BIH ;
BITMAP bmpInfo;
BITMAPFILEHEADER bfh;

////////////////////////////////////////////////
 int iSize = sizeof(BIH->bmiHeader) + 256*sizeof(RGBQUAD) ;
  BIH = (BITMAPINFO *)LocalAlloc( LPTR, iSize );
   memset(BIH, 0, iSize);
```

```
 // Fill in the header info.
 BIH->bmiHeader.biSize = sizeof(BIH->bmiHeader);
BIH->bmiHeader.biWidth = width1;
 BIH->bmiHeader.biHeight = height1;
 BIH->bmiHeader.biPlanes = 1;
 BIH->bmiHeader.biBitCount = 8;
 BIH->bmiHeader.biCompression = BI_RGB;
for(i = 0; i < 256; i++)
{
  BIH->bmiColors[i].rgbBlue  = i;
  BIH->bmiColors[i].rgbGreen = i;
  BIH->bmiColors[i].rgbRed   = i;
  BIH->bmiColors[i].rgbReserved = 0;
}

// Create a new DC.
 m_hdc = ::CreateCompatibleDC(NULL) ;


 // Create the DIB section.
 hBitmap = CreateDIBSection(   m_hdc,
                       BIH,
                       DIB_RGB_COLORS,
                       &m_pBits,
                       NULL,
                       0);



 memcpy(bitmapArray, p,(height1*(width1+padding)));

 iSize = SetBitmapBits(hBitmap, (width1)*height1, bitmapArray);
CBitmap::FromHandle(hBitmap)->GetBitmap(&bmpInfo);
 bmpInfo.bmHeight = height1;
 bmpInfo.bmWidth  = width1;




      ULONG sizBMI;

          BIH->bmiHeader.biSizeImage=BIH->bmiHeader.biWidth*BIH-
>bmiHeader.biHeight*(BIH->bmiHeader.biBitCount/8);

       //finding the sizebmi variable
      sizBMI = sizeof(BITMAPINFOHEADER)+sizeof(RGBQUAD)*(1<<BIH-
>bmiHeader.biBitCount);

       //creating bimap fileheader info
       bfh.bfType = 0x4D42;
```

```
        bfh.bfSize =
sizeof(BITMAPFILEHEADER)+sizeof(BITMAPINFOHEADER)+sizBMI+   BIH-
>bmiHeader.biSizeImage;
        bfh.bfReserved1 =    bfh.bfReserved2 = 0;
        bfh.bfOffBits = sizeof(BITMAPFILEHEADER)+sizBMI;


       LocalFree(BIH);

       ReleaseDC(NULL, m_hdc);
       return hBitmap;

}

void camera::release()
{
//release buffers
MbufFree(GreyImage1);
MbufFree(GreyImage2);

MappFreeDefault(MilApplication, MilSystem, MilDisplay,
                        MilDigitizer, M_NULL);


delete [] userImage1;
delete [] userImage2;
delete [] bitmapArray;

}



void camera::export()
{
//save obtained images for debugging puprposes
FILE* fil;
char t[20];
char tx[20];
char ff[20];
int f=0;
do {
  if(f) fclose(fil);
  f++;
  _itoa(f,ff,10);
  strcpy(t,"images\\right-");
  strcat(t,ff);
  strcat(t,".bmp");
    strcpy(tx,"images\\left-");
  strcat(tx,ff);
  strcat(tx,".bmp");
  fil=fopen(t,"r");
} while(fil);
```

```
MbufExport(t,M_BMP ,GreyImage1);
MbufExport(tx,M_BMP ,GreyImage2);


}


void camera::export2()
{
//save obtained images for debugging puprposes
FILE* fil;
char t[20];
char tx[20];
char ff[20];
int f=0;
do {
  if(f) fclose(fil);
  f++;
  _itoa(f,ff,10);
  strcpy(t,"target\\right-");
  strcat(t,ff);
  strcat(t,".bmp");
    strcpy(tx,"target\\left-");
  strcat(tx,ff);
  strcat(tx,".bmp");
  fil=fopen(t,"r");
} while(fil);

MbufExport(t,M_BMP ,GreyImage1);//these functions are from a third party commercial library
//designed for interfacing with cameras matrox mil library)
MbufExport(tx,M_BMP ,GreyImage2);


}

void camera::GRAB()
{
//clear the bugffers
MbufClear(GreyImage1, 0x0); MbufClear(GreyImage2, 0x0);
//start grabbing from channel 1
MdigChannel(MilDigitizer,M_CH0);
MdigGrab(MilDigitizer,  GreyImage1);MdigGrab(MilDigitizer,  GreyImage1);
//start grabbing from channel 2
MdigChannel(MilDigitizer,M_CH1);
MdigGrab(MilDigitizer,  GreyImage2);MdigGrab(MilDigitizer,  GreyImage2);
MbufGet2d(GreyImage1,0,0,S_X,S_Y,userImage1);
MbufGet2d(GreyImage2,0,0,S_X,S_Y,userImage2);

}

void camera::initialize(void)
```

```
{

MappAllocDefault(M_SETUP, &MilApplication, &MilSystem,&MilDisplay, &MilDigitizer,
M_NULL);
S_Y=MdigInquire(MilDigitizer, M_SIZE_Y   , M_NULL);
S_X=MdigInquire(MilDigitizer, M_SIZE_X   , M_NULL);
//i=S_Y
MbufAlloc2d(MilSystem,S_X,S_Y,8+M_UNSIGNED,M_IMAGE+M_PROC+M_GRAB+M_
DISP,&GreyImage1);
MbufAlloc2d(MilSystem,S_X,S_Y,8+M_UNSIGNED,M_IMAGE+M_PROC+M_GRAB+M_
DISP,&GreyImage2);


userImage1 = new unsigned char[S_X*S_Y];
userImage2 = new unsigned char[S_X*S_Y];


MdigControl(MilDigitizer, M_GRAB_DIRECTION_X, M_REVERSE);
MdigControl(MilDigitizer, M_GRAB_DIRECTION_Y, M_REVERSE);
//MdispSelect(MilDisplay, GreyImage1);
 padding = 0;
while ( (S_X + padding ) % 4 != 0 )
    padding++;

  bitmapArray = new BYTE[(S_X+padding)*S_Y];


}


void camera::border(MPoint* pts,CBitmap *CBT, CBitmap *CBC)
{

  short*flags = new short[S_X*S_Y];
  register int i,j;
  int flag2=0;

  int relative=0;
  int n=0;
  int obj,bord;
  int cc_border=0,cc_circles=0;
  BYTE* pBits;
  BYTE* cBits;
  BITMAP bm;
  for (i = 0; i < S_X*S_Y; i++) {flags[i]=255;}//intlizae the flags
  COR this_b,previous_b;
  COR neighbours2[10];
  COR borders[11][70];
  COR2 centres[10];
  int numborders[140];
for(i=0;i<140;i++) numborders[i]=0;
  CBT->GetBitmap(&bm);
```

```
  pBits= (BYTE*) bm.bmBits +(S_Y-1)*(S_X);
  cBits= (BYTE*) bm.bmBits ;

  bord=0;
  obj=0;
  for (i = 0; i < S_Y; i++){
   for (j = 0; j < S_X; j++)
  {
    if(i!=(S_Y-1))
  if(pBits[0]>pBits[-1] && (flags[(i*S_X)+j]==255))//detect first element of region
  {
    if(obj==0 || (numborders[obj]>45 && numborders[obj]<100  && (
variance(borders[obj],numborders[obj])<1)) )
      obj++;


    numborders[obj]=0;
    bord=0;
    this_b.x=j;this_b.y=i;
    previous_b.x=j-1;previous_b.y=i;
    borders[obj][bord]=this_b;

    //cant detect any neighbours
    while(1){
      if(flags[(this_b.y*S_X)+this_b.x]==0 ) break;
      else flags[(this_b.y*S_X)+this_b.x]=0;

  findneighbours(neighbours2,previous_b,this_b);
    for(n=2;n<=8;n++){

      if(cBits[((S_Y-neighbours2[n].y-1)*S_X)+neighbours2[n].x]>250){
        numborders[obj]++;
        previous_b=this_b;
        this_b=neighbours2[n];

        bord++;

      borders[obj][bord]=neighbours2[n];
      break;
      }
    }
    } }


  pBits++;
  }
 pBits=pBits-2*S_X;
}

  if(numborders[obj]<30) {numborders[obj]=0; obj--;}

CBC->GetBitmap(&bm);
```

```
pBits= (BYTE*) bm.bmBits +(S_Y-1)*(S_X);
  for (i = 0; i < S_X*S_Y; i++) {flags[i]=255;}


long double xxx=0,yyy=0;


for (i = 1; i <= obj; i++){
  xxx=0;yyy=0;
    for (j = 0; j < numborders[i]; j++)
  {
xxx+=borders[i][j].x;
yyy+=borders[i][j].y;


}
  xxx/=(double)numborders[i];
  yyy/=(double)numborders[i];
  yyy+=1;
centres[i].x=xxx;
centres[i].y=yyy;

}
for (i = 1; i <= obj; i++){flags[(((int)centres[i].y)*S_X)+((int)centres[i].x)]=0;}

for (i = 0; i < S_Y; i++){
   for (j = 0; j < S_X; j++)
  {
    if(flags[(i*S_X)+j]==0)
    (*pBits)=0;

  pBits++;
  }
 pBits=pBits-2*S_X;
}

  CBT->GetBitmap(&bm);
  pBits= (BYTE*) bm.bmBits +(S_Y-1)*(S_X);

for (i = 0; i < S_Y; i++){
   for (j = 0; j < S_X; j++)
  {

    if(*pBits==0)
      (*pBits)=255;
    else (*pBits)=0;

  pBits++;
  }
 pBits=pBits-2*S_X;
}
delete [] flags;
```

```
for(i=1;i<=10;i++){

pts[i].x=centres[i].x;
pts[i].y=centres[i].y;
}

}
void camera::findneighbours(COR* result, COR prev, COR curn)
{
int i;
int xx,yy;
int relative;
xx=prev.x-curn.x;
yy=prev.y-curn.y;

for(i=0;i<=9;i++){result[i].x=0;result[i].y=0;}

for(i=0;i<=7;i++){
 if(neighbour1[i].x==xx && neighbour1[i].y==yy)
 {
  relative=i;
  break;
 }
}
int ftemp;
for(i=1;i<=8;i++){
 ftemp=(relative+i)%8;
 result[i+1].x=curn.x+ neighbour1[(relative+i)%8].x  ;
 result[i+1].y=curn.y+  neighbour1[(relative+i)%8].y  ;
}
}

void camera::threshold1(CBitmap* CB)
{
 int i,j;
 BITMAP bm;
 BYTE* pBits;
 double temp=0;
 int t_old=0,t_new=125;
 CB->GetBitmap(&bm);
 pBits= (BYTE*) bm.bmBits +(S_Y-1)*(S_X);//no padding required
 double* hist = new double[256];
 int threshold=0;
 //note this program might not work with a camera whose image width is not a multiple //of 4
or an RGB camera , all the code is written for grayscale images
for (i = 0; i < 256; i++) hist[i]=0;
//create histogram
for (i = 0; i < S_Y; i++){
   for (j = 0; j < S_X; j++)
 {
 hist[*(pBits)]++;
```

```
  pBits++;
  }
 pBits=pBits-2*S_X;
}

for (i = 0; i < 256; i++) hist[i]/=(S_X*S_Y);//divide by total pixels to get probability


while(t_old!=t_new){
  t_old=t_new;
  temp=(SX(hist,t_old)/S(hist,t_old))+((SX(hist,255)-SX(hist,t_old))/(S(hist,255)-S(hist,t_old)));
  temp*=0.5;
  t_new=(int)temp;
  }
t_new+=40;

pBits= (BYTE*) bm.bmBits +(S_Y-1)*(S_X);//necessary for rewindidng the pointer
for (i = 0; i < S_Y; i++){
   for (j = 0; j < S_X; j++)
  {
  if((*(pBits))>t_new) *(pBits)=255;
  else *(pBits)=0;

  pBits++;
  }
 pBits=pBits-2*S_X;
}

delete [] hist;
}
double camera::SX(double *a, int t)
{
int k=0;
double temp=0;
for(k=0;k<=t;k++)
temp+=a[k]*k;
return temp;
}

double camera::S(double *a, int t)
{
int k=0;
double temp=0;
for(k=0;k<=t;k++)
temp+=a[k];

return temp;
}

void camera::getworldpos(MPoint *pts,  int plane_num)
{
  int j=0,i=0;
```

```
  double x1,y1,z1;
    //assume 5 CM distance
  double x_interval=2.5;//distance seperating disks on the control surface
  double y_interval=2.5;//same as above for the y axis


  char* ctemp=new char[10];
  string temp1="world//pos";
  itoa(plane_num, ctemp, 10 );
  string temp3=temp1+ctemp+".txt";
  fstream world_file(temp3.c_str());

  world_file >> z1;
  world_file >> x1;
  world_file >> y1;

  pts[1].x=x1;
  pts[1].y=y1;
  pts[1].z=z1;

//dummy values
  pts[10].x=0;
  pts[10].y=0;
  pts[10].z=0;



  for(i=1;i<=3;i++)
    for(j=1;j<=3;j++)
      if(i+j!=2) {
        pts[((i-1)*3)+j].x=x1+(j-1)*x_interval;
        pts[((i-1)*3)+j].y=y1+(i-1)*y_interval;
        pts[((i-1)*3)+j].z=0;
            }



  delete [] ctemp;
}


void camera::normalize1(MPoint* pts){

int i;
//this function will simply divide the coordinates to give the values
//relative to a centre (y/2, x/2)
for(i=1;i<=9;i++){
pts[i].x-=(double)S_X/2;
pts[i].y-=(double)S_Y/2;

}
```

```
}
void camera::getworldpos2(MPoint pts[11],double x1,double y1,double z1){

  int j=0,i=0;

  double x_interval=2.5;//distance seperating disks on the control surface
  double y_interval=2.5;//same as above for the y axis

  pts[1].x=x1;
  pts[1].y=y1;
  pts[1].z=z1;

 //dummy values
  pts[10].x=0;
  pts[10].y=0;
  pts[10].z=0;


  for(i=1;i<=3;i++)
    for(j=1;j<=3;j++)
      if(i+j!=2) {
        pts[((i-1)*3)+j].x=x1+(i-1)*x_interval;
        pts[((i-1)*3)+j].y=y1+(j-1)*y_interval;
        pts[((i-1)*3)+j].z=z1;
            }

 }
double camera::variance(COR *A, int N)
{
int j;

double ct=0;

long double xxx=0,yyy=0;
double xx,yy;

  xxx=0;yyy=0;
   for (j = 0; j < N; j++)
  {
xxx+=A[j].x;
yyy+=A[j].y;
}
  xxx/=(double)N;
  yyy/=(double)N;
  yyy+=1;
xx=xxx;
yy=yyy;

double* t=new double[N+1];
  xxx=0;yyy=0;
   for (j = 0; j < N; j++)
  {
```

```
xxx=(A[j].x-xx)*(A[j].x-xx)+(A[j].y-yy)*(A[j].y-yy);
xxx=sqrt(xxx);
t[j]=xxx;
yyy+=xxx;

}

//mean

yyy=yyy/N;
xxx=0;
double zz;
   for (j = 0; j < N; j++)
  {
zz=(t[j]-yyy);

xxx+=(zz*zz);
}

xxx/=N;

delete [] t;
return xxx;
}

void camera::normalize_image(double** left,double** right,double** world, CENTRES* ct)
{
int i,j;
double X_mean;
double Y_mean;
double Z_mean;
double distance_mean;
double d_temp;
double N=27.000;
double X,Y,Z;
double coeff=0;

//find for left camera
Z_mean=d_temp=X_mean=Y_mean=distance_mean=0;


for(j=1;j<=3;j++)
for(i=1;i<=9;i++){
X=ct[j].LPoints[i].x;
Y=ct[j].LPoints[i].y;

X_mean+=X;
Y_mean+=Y;
}
//found centroid
X_mean/=N;
Y_mean/=N;
```

```
for(j=1;j<=3;j++)
for(i=1;i<=9;i++){
X=ct[j].LPoints[i].x-X_mean;
Y=ct[j].LPoints[i].y-Y_mean;
d_temp+=sqrt((X*X)+(Y*Y));
}
//found distance mean
distance_mean=d_temp/N;
coeff=sqrt((double)2.0000); coeff=coeff/distance_mean;// coeff=sqrt(coeff);
X_mean=((double)-1)*X_mean*coeff;
Y_mean=((double)-1)*Y_mean*coeff;

//now mnormalize left camera

for(j=1;j<=3;j++)
for(i=1;i<=9;i++){
ct[j].LPoints[i].x=(ct[j].LPoints[i].x*coeff)+X_mean;
ct[j].LPoints[i].y=(ct[j].LPoints[i].y*coeff)+Y_mean;
}

//now form the normalization matrix
left[1][1]=left[2][2]=coeff;
left[1][3]=X_mean;
left[2][3]=Y_mean;
left[3][3]=1;
//find for right camera
Z_mean=d_temp=X_mean=Y_mean=distance_mean=0;


for(j=1;j<=3;j++)
for(i=1;i<=9;i++){
X=ct[j].RPoints[i].x;
Y=ct[j].RPoints[i].y;
X_mean+=X;
Y_mean+=Y;
}

X_mean/=N;
Y_mean/=N;
for(j=1;j<=3;j++)
for(i=1;i<=9;i++){
X=ct[j].RPoints[i].x-X_mean;
Y=ct[j].RPoints[i].y-Y_mean;
d_temp+=sqrt((X*X)+(Y*Y));
}


distance_mean=d_temp/N;
coeff=sqrt((double)2.0000); coeff=coeff/distance_mean;
X_mean=((double)-1)*X_mean*coeff;
Y_mean=((double)-1)*Y_mean*coeff;
```

```
for(j=1;j<=3;j++)
for(i=1;i<=9;i++){
ct[j].RPoints[i].x=ct[j].RPoints[i].x*coeff+X_mean;
ct[j].RPoints[i].y=ct[j].RPoints[i].y*coeff+Y_mean;
}

//now form the normalization matrix
right[1][1]=right[2][2]=coeff;
right[1][3]=X_mean;
right[2][3]=Y_mean;
right[3][3]=1;


//find for world coordinates
Z_mean=d_temp=X_mean=Y_mean=distance_mean=0;


for(j=1;j<=3;j++)
for(i=1;i<=9;i++){
X=ct[j].WPoints[i].x;
Y=ct[j].WPoints[i].y;
Z=ct[j].WPoints[i].z;
X_mean+=X;
Y_mean+=Y;
Z_mean+=Z;

}
X_mean/=N;
Y_mean/=N;
Z_mean/=N;


for(j=1;j<=3;j++)
for(i=1;i<=9;i++){
X=ct[j].WPoints[i].x-X_mean;
Y=ct[j].WPoints[i].y-Y_mean;
Z=ct[j].WPoints[i].z-Z_mean;
d_temp+=sqrt((X*X)+(Y*Y)+(Z*Z));

}

distance_mean=d_temp/N;
coeff=sqrt((double)3.0000); coeff=coeff/distance_mean;// coeff=sqrt(coeff);
X_mean=((double)-1)*X_mean*coeff;
Y_mean=((double)-1)*Y_mean*coeff;
Z_mean=((double)-1)*Z_mean*coeff;

for(j=1;j<=3;j++)
for(i=1;i<=9;i++){
ct[j].WPoints[i].x=ct[j].WPoints[i].x*coeff+X_mean;
ct[j].WPoints[i].y=ct[j].WPoints[i].y*coeff+Y_mean;
```

```
ct[j].WPoints[i].z=ct[j].WPoints[i].z*coeff+Z_mean;
}

//no form the normalization matrix
world[1][1]=world[2][2]=world[3][3]=coeff;
world[1][4]=X_mean;
world[2][4]=Y_mean;
world[3][4]=Z_mean;
world[4][4]=1;
}
void camera::denormalize(double** leftP, double** rightP, double **left, double **right, double
**world)
{
//find the inverse of the normalization matrix for the two cameras
// srtarting from the left camera
double deter;
double mat[5][5];
double temp[5][5];
int i,j;

deter=leftP[1][1]; deter=deter*deter; deter=((double)1)/deter;

//finding the matrix inverse

//initliaze the matrix
for(i=1;i<=3;i++)
for(j=1;j<=3;j++){
mat[i][j]=0;
temp[i][j]=0;
}


mat[1][1]=leftP[1][1];
mat[1][3]=((double)-1)*leftP[1][1]*leftP[1][3];
mat[2][2]=leftP[1][1];
mat[2][3]=((double)-1)*leftP[1][1]*leftP[2][3];
mat[3][3]=leftP[1][1]*leftP[1][1];

//multiply in the determinant
for(i=1;i<=3;i++)
for(j=1;j<=3;j++)
mat[i][j]*=deter;


for(i=1;i<=3;i++)
for(j=1;j<=4;j++)
{
temp[i][j]=mat[i][1]*left[1][j]+mat[i][2]*left[2][j]+mat[i][3]*left[3][j];

}

//multiply again on the right side by the world normalization matrix
```

```
for(i=1;i<=3;i++)
for(j=1;j<=4;j++)
{
left[i][j]=temp[i][1]*world[1][j]+temp[i][2]*world[2][j]+temp[i][3]*world[3][j]+temp[i][4]*world[4][j
];
}
for(i=1;i<=3;i++)
for(j=1;j<=4;j++)
{
left[i][j]/=left[3][4]  ;
}
//now the right camera

for(i=1;i<=3;i++)
for(j=1;j<=3;j++){
mat[i][j]=0;
temp[i][j]=0;
}
mat[1][1]=rightP[1][1];
mat[1][3]=((double)-1)*rightP[1][1]*rightP[1][3];
mat[2][2]=rightP[1][1];
mat[2][3]=((double)-1)*rightP[1][1]*rightP[2][3];
mat[3][3]=rightP[1][1]*rightP[1][1];

//multiply in the determinant
for(i=1;i<=3;i++)
for(j=1;j<=3;j++)
mat[i][j]*=deter;


for(i=1;i<=3;i++)
for(j=1;j<=4;j++)
{
temp[i][j]=mat[i][1]*right[1][j]+mat[i][2]*right[2][j]+mat[i][3]*right[3][j];
}

//multiply again on the right side by the world normalization matrix
for(i=1;i<=3;i++)
for(j=1;j<=4;j++)
{
right[i][j]=temp[i][1]*world[1][j]+temp[i][2]*world[2][j]+temp[i][3]*world[3][j]+temp[i][4]*world[4]
[j];
}

for(i=1;i<=3;i++)
for(j=1;j<=4;j++)
{
right[i][j]/=right[3][4]  ;

}
}
```

```
void camera::finderror(double* leftx, double* lefty, double* rightx, double* righty, double **PL,
double **PR, CENTRES *ct)
{
int i,j;
*leftx=*lefty=*rightx=*righty=0;//initliaze
double reprojection_x;
double reprojection_y;
double reprojection_w;
double errx,erry;

//printing the errors
FILE *lfile,*rfile;
lfile=fopen("lefterr.txt","wt");
rfile=fopen("refterr.txt","wt");
fprintf(lfile,"xerror yerror --  X   Y\n\n");
fprintf(rfile,"xerror yerror --  X   Y\n\n");

for(j=1;j<=3;j++)
for(i=1;i<=9;i++){

//left camera
reprojection_w=PL[3][1]*ct[j].WPoints[i].x+PL[3][2]*ct[j].WPoints[i].y+PL[3][3]*ct[j].WPoints[i].z
+PL[3][4];
reprojection_x=PL[1][1]*ct[j].WPoints[i].x+PL[1][2]*ct[j].WPoints[i].y+PL[1][3]*ct[j].WPoints[i].z
+PL[1][4];
reprojection_y=PL[2][1]*ct[j].WPoints[i].x+PL[2][2]*ct[j].WPoints[i].y+PL[2][3]*ct[j].WPoints[i].z
+PL[2][4];

//last coefficient has to be one
reprojection_x/=reprojection_w;
reprojection_y/=reprojection_w;

errx=fabs(ct[j].LPoints[i].x-reprojection_x);//*(ct[j].LPoints[i].x-reprojection_x);
erry=fabs(ct[j].LPoints[i].y-reprojection_y);//*(ct[j].LPoints[i].y-reprojection_y);

fprintf(lfile,"%0.3f   %0.3f  ---  %0.3f   %0.3f -- LEVEL=%d
POINT=%d\n",errx,erry,ct[j].LPoints[i].x,ct[j].LPoints[i].y,j,i);

*leftx+=errx;
*lefty+=erry;

//now right camera
reprojection_w=PR[3][1]*ct[j].WPoints[i].x+PR[3][2]*ct[j].WPoints[i].y+PR[3][3]*ct[j].WPoints[i].
z+PR[3][4];
reprojection_x=PR[1][1]*ct[j].WPoints[i].x+PR[1][2]*ct[j].WPoints[i].y+PR[1][3]*ct[j].WPoints[i].z
+PR[1][4];
reprojection_y=PR[2][1]*ct[j].WPoints[i].x+PR[2][2]*ct[j].WPoints[i].y+PR[2][3]*ct[j].WPoints[i].z
+PR[2][4];

//last coefficient has to be one
reprojection_x/=reprojection_w;
reprojection_y/=reprojection_w;
```

```
errx=fabs(ct[j].RPoints[i].x-reprojection_x);//*(ct[j].RPoints[i].x-reprojection_x);
erry=fabs(ct[j].RPoints[i].y-reprojection_y);//*(ct[j].RPoints[i].y-reprojection_y);

fprintf(rfile,"%0.3f  %0.3f --- %0.3f  %0.3f -- LEVEL=%d  POINT=%d
\n",errx,erry,ct[j].RPoints[i].x,ct[j].RPoints[i].y,j,i);

*rightx+=errx;
*righty+=erry;
}

fclose(lfile);
fclose(rfile);

*rightx/=((double)27)  ;
*righty/=((double)27);
*leftx/=((double)27)  ;
*lefty/=  ((double)27);
}

void camera::mrqmin(double x[], double y[], double sig[], int ndata, double a[], int ia[], int ma,
double **covar, double **alpha, double *chisq, void (__cdecl *funcs)(double,double [],double
*,double [],int), double *alamda)
{
}

void camera::find_center(double **P, double *C)
{
//second implementation of centre finding algorithm
//note this is the centre of projection such that PC=0
//SVD is used

int i,j;
double** u;
u= new double* [7];
 for (i=0; i<7; i++)
{ u[i] = new double[6];

 }

 //fill u with A
for (i=1; i<=3; i++)
for (j=1; j<=4; j++){

u[i][j]=P[i][j];

}
//appending last rows as zeros for SVD its recommended
// to have at least as much rows as columns

for (j=1; j<=4; j++){
u[4][j]=0;
```

```
}

//do SVD
solve2(u, 4, 4,C);

for (j=1; j<=4; j++) C[j]/=C[4];

FILE* tempe;
if(flag3!=4)
{tempe=fopen("leftC.txt","wt"); flag3=3;}


if(flag3==4)
tempe=fopen("rightC.txt","wt");

if(flag3==3) flag3=4;


fprintf(tempe,"C=[ %0.5f ; %0.5f  ; %0.5f ; %0.5f]",C[1],C[2],C[3],C[4]);
fclose(tempe);

for (i=0; i<7; i++)
{ delete [] u[i];

}
 delete [] u;
}

void camera::findF(double** FL,double** FR, CENTRES* ct)
{
int i,j;

//first find the pseudo inverse of the projection matrix

double** u;

double* x=new double[15];
//creating temporary matrices
u= new double* [37];

 for (i=0; i<37; i++)
 u[i] = new double[16];

//find F for left camera, form A for SVD
 int cc=1;

for (j=1; j<=3; j++)
for (i=1; i<=9; i++)
{u[cc][1]=ct[j].RPoints[i].x*ct[j].LPoints[i].x;
u[cc][2]=ct[j].RPoints[i].x*ct[j].LPoints[i].y;
u[cc][3]=ct[j].RPoints[i].x;
u[cc][4]=ct[j].RPoints[i].y*ct[j].LPoints[i].x;
```

```
u[cc][5]=ct[j].RPoints[i].y*ct[j].LPoints[i].y;
u[cc][6]=ct[j].RPoints[i].y;
u[cc][7]=ct[j].LPoints[i].x;
u[cc][8]=ct[j].LPoints[i].y;
u[cc][9]=1;
cc++;
}
writematrix(u,27,9,"ULEFT.txt");
//now do svdcmp
solve2(u,27,9,x);


cc=1;
for (j=1; j<=3; j++)
for (i=1; i<=3; i++) {
FL[j][i]=x[cc]; cc++;
}
///now right F
for (j=1; j<=3; j++)
for (i=1; i<=9; i++)
{u[cc][1]=ct[j].LPoints[i].x*ct[j].RPoints[i].x;
u[cc][2]=ct[j].LPoints[i].x*ct[j].RPoints[i].y;
u[cc][3]=ct[j].LPoints[i].x;
u[cc][4]=ct[j].LPoints[i].y*ct[j].RPoints[i].x;
u[cc][5]=ct[j].LPoints[i].y*ct[j].RPoints[i].y;
u[cc][6]=ct[j].LPoints[i].y;
u[cc][7]=ct[j].RPoints[i].x;
u[cc][8]=ct[j].RPoints[i].y;
u[cc][9]=1;
cc++;
}

writematrix(u,27,9,"URIGHT.txt");
//now do svdcmp
solve2(u,27,9,x);

cc=1;
for (j=1; j<=3; j++)
for (i=1; i<=3; i++) {
FR[j][i]=x[cc]; cc++;
}


//now write the two Fs to file
writematrix(FL,3,3,"left_f.txt");
writematrix(FR,3,3,"right_f.txt");


//freeing temporary arrays
 for (i=0; i<37; i++)
{ delete [] u[i];
```

```
}
 delete [] u;
}
void camera::writematrix(double **x,int m,int n, char *name)
{

int i,j;
FILE* tempf;


tempf=fopen(name,"wt");
  //fprintf(tempf,"M=[ ");
  for(i=1;i<=m;i++){
  // if(i!=1) fprintf(tempf," ;");
    for(j=1;j<=n;j++){
      fprintf(tempf,"%0.9f  ",x[i][j]);
  }}

fprintf(tempf,"  ");
fclose(tempf);
}
void camera::writevector(double *x,int n, char *name)
{

int j;
FILE* tempf;
tempf=fopen(name,"wt");

  fprintf(tempf,"m=[ ");

    for(j=1;j<=n;j++){
       if(j!=1) fprintf(tempf," ; ");
      fprintf(tempf,"%0.7f ",x[j]);
           }

fprintf(tempf," ] ");
fclose(tempf);
}

void camera::matrixmultiply(double **c, double **a, double **b, int ma, int na, int mb, int nb)
{//C=AxB    ma is number of rows of a, na is the number of columns of a, mb and nb is same
for b
  if(na!=mb) {AfxMessageBox("Matrix multiplication: size error"); return;}
int i,j,k;
double temp;
double** H;//this matrix is used as a temporary placeholder so we could have c=c*a
H= new double* [ma+2];
 for (i=0; i<ma+2; i++)
H[i] = new double[nb+2];

for (i=1; i<=ma; i++)
for (j=1; j<=nb; j++){
```

```
temp=0;
for (k=1; k<=na; k++)
  temp+=a[i ][k] *b[k][j];

H[i][j]=temp;
}

for (i=1; i<=ma; i++)
for (j=1; j<=nb; j++)
c[i][j]=H[i][j];//now put the results back in C


//free the temprorary matrix
for (i=0; i<ma+2; i++)
delete [] H[i];

delete [] H;
}

void camera::solve2(double **u, int m, int n,  double *x)
{
int j,i;
double** v;
double* w=new double[m+2];
v= new double* [m+2];
 for (i=0; i<m+2; i++)
 v[i] = new double[n+2];
for(i=0;i<=m;i++)
for(j=0;j<=n;j++)
v[i][j]=0;
for(i=0;i<=m;i++) w[i]=0;
svd2(u,m,n,w,v);
int smallest_index=0;
double smallest_value=200000;
  for (j = 1; j <= n; j++){
    if(w[j]<smallest_value) {smallest_value=w[j]; smallest_index=j;}
  }
  for (j = 1; j <= n; j++){
x[j]=v[j][smallest_index];

  }
for (i=0; i<m+2; i++)
delete [] v[i];
delete v;
delete [] w;



}

void camera::svd2(double **A, int m, int n, double *w, double **v)
{
```

```
{//code taken from Numerical Recipes in C , will not be shown here
 }
}
void camera::getworldpos3(double* x1,double* y1,double* z1,int plane)
{
  int j=0,i=0;
  char* ctemp=new char[10];
  string temp1="world//pos";
  itoa(plane, ctemp, 10 );
  string temp3=temp1+ctemp+".txt";
  fstream world_file(temp3.c_str());

  world_file >> *x1;
  world_file >> *y1;
  world_file >> *z1;

  delete [] ctemp;

}
```

// camera.h: interface for the camera class.

```
//
//////////////////////////////////////////////////////////////////////
#include "mil.h"

  typedef struct {
        double x;
       double y;
      double z;

    } MPoint;

    typedef struct {
        MPoint LPoints[12];
       MPoint RPoints[12];
      MPoint WPoints[12];

  } CENTRES;
#if
!defined(AFX_CAMERA_H__8CA0399E_1EC2_475B_8927_AC79161D81C4__INCLUDED_
)
#define AFX_CAMERA_H__8CA0399E_1EC2_475B_8927_AC79161D81C4__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

class camera
{
public:
  void svd2(double **A, int m, int n, double *w, double **v);
  void solve2(double **u, int m, int n,  double *x);
```

```
  void matrixmultiply(double** c,double** a,double** b, int ma,int na, int mb, int nb);
  void writevector(double* x,int n,char* name);
  void writematrix(double** x,int m,int n,char* name);
  void findF(double** FL,double** FR, CENTRES* ct);
  void find_center(double **P, double *C);
  void finderror(double* leftx,double* lefty,double* rightx,double* righty,  double** PL,double**
PR,CENTRES* ct);
  void denormalize(double** leftP, double** rightP,double** left, double** right, double** world);
  void normalize_image(double **left,double **right,double** world, CENTRES* ct);
  void getworldpos3(double* x1,double* y1,double* z1,int plane);
  void mrqmin(double x[], double y[], double sig[], int ndata, double a[], int ia[],int ma, double
**covar, double **alpha, double *chisq,void (*funcs)(double, double [], double *, double [], int),
double *alamda);
  void getworldpos2(MPoint pts[11],double x1,double y1,double z1);
  int padding;

    typedef struct {
       int x;
       int y;
     //int k;

    public:

  } COR;

       typedef struct {
       double x;
       double y;
     //int k;

  } COR2;

  double variance(COR* A,int N);

  void getworldpos(MPoint pts[11],int plane_num);
  void border(MPoint* pts,CBitmap *CBT, CBitmap *CBC);
  void normalize1(MPoint* pts);
  double S(double* a,int t);
  double SX(double* a,int t);
  void threshold1(CBitmap* CB);

  HBITMAP ArrayToBitmap(unsigned char *p);
  void export(void);
  void export2(void);
  void release(void);


  void GRAB(void);
  int S_X;
  int S_Y;
int flag3;
int flag4;
```

```
int flag5;

  unsigned char* userImage1;
  unsigned char* userImage2;
// MIL_ID MilApplication,MilSystem, MilDisplay,MilDigitizer,
MilImage,GreyImage1,GreyImage2;

    void initialize(void);
  camera();
  virtual ~camera();

private:

  BYTE*bitmapArray;
  MIL_ID MilApplication,MilSystem, MilDisplay,MilDigitizer,
MilImage,GreyImage1,GreyImage2;

  COR neighbour1[10];
  void findneighbours(COR* result,COR prev,COR curn);
};

#endif //
!defined(AFX_CAMERA_H__8CA0399E_1EC2_475B_8927_AC79161D81C4__INCLUDED_
)
```

////////////////// Order.cpp : implementation file////////
//COrder dialog This is a GUI class for conveying calibration data o the
//user and also interacting with the user regarding the order of the
//calibration targets

```
#include "stdafx.h"
#include "calibrb.h"
#include "Order.h"
#include "math.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endifCOrder::COrder(CWnd* pParent /*=NULL*/)
  : CDialog(COrder::IDD, pParent)
{
  //{{AFX_DATA_INIT(COrder)
  m_x = 0.0;
  m_y = 0.0;
  m_z = 0.0;
  //}}AFX_DATA_INIT
  completion_flag=0;
  int i;
  for(i=1;i<=9;i++){
```

```
    rect_highlight[i]=0;
    rect_highlight2[i]=0;
  order[i]=0;
  }

flagtimer=0;
model[1].x=672;
model[1].y=451;
colorflag=0;
prev_circle=0;
count=0;
prev_point.x=0;
prev_point.y=0;
value_set_flag2=0;
}


void COrder::DoDataExchange(CDataExchange* pDX)
{
  CDialog::DoDataExchange(pDX);
  //{{AFX_DATA_MAP(COrder)
  DDX_Text(pDX, IDC_EDIT1, m_x);
  DDX_Text(pDX, IDC_EDIT2, m_y);
  DDX_Text(pDX, IDC_EDIT3, m_z);
  //}}AFX_DATA_MAP
}


BEGIN_MESSAGE_MAP(COrder, CDialog)
  //{{AFX_MSG_MAP(COrder)
  ON_WM_PAINT()
  ON_BN_CLICKED(IDDONT, OnDont)
  ON_WM_LBUTTONDOWN()
  ON_WM_MOUSEMOVE()
  ON_WM_TIMER()
  ON_BN_CLICKED(IDRESET, OnReset)
  ON_BN_CLICKED(IDC_BUTTON1, OnButton1)
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
//
// COrder message handlers

void COrder::OnCancel()
{
MessageBox("You must choose the order","Calibration");
  CDialog::OnCancel();
}

void COrder::loadimage(MPoint *pts, CBitmap *bt,int width, int height,LPCTSTR lpszString,int
value_set_flag,double* x,double* y,double* z)
```

```
{
ptspts=pts;
btbt=bt;
w= width ;
h= height ;
if(value_set_flag==1){
m_x=*x;
m_y=*y;
m_z=*z;

}
if(value_set_flag==0){
m_x=21.20;
m_y=34.65;
m_z=0;
  }

X=x;Y=y;Z=z;

value_set_flag2=value_set_flag;



  myrect.left=20;
  myrect.top= 90 ;
  myrect.right= 500 ;
  myrect.bottom= (( myrect.right-myrect.left) *h)/w+myrect.top;
int i,j;
double width2=(double)myrect.Width();
double height2=(double)myrect.Height();
double x_offset=(double)myrect.left    ;
double y_offset=(double)    myrect.top         ;
double hh=h;
double ww=w;
int diameter =10;


for(i=1;i<=9;i++)
{
temp_points[i].x=(long)((ptspts[i].x)*((double)width2/ww)+(double)x_offset);
temp_points[i].y=(long)((ptspts[i].y)*((double)height2/hh)+(double)y_offset);


circle_rect[i].left=temp_points[i].x-diameter   ;
circle_rect[i].top= temp_points[i].y+ diameter ;
circle_rect[i].right= temp_points[i].x+diameter;
circle_rect[i].bottom=temp_points[i].y-diameter ;

}

model_rect.left=540  ;
model_rect.top= 280  ;
```

```
model_rect.right=740;
model_rect.bottom= 550;

//find coordinates of model plane

int x_interval=42;
int y_interval=59;
  for(i=1;i<=3;i++)
    for(j=1;j<=3;j++)
      if(i+j!=2) {
        model[((i-1)*3)+j].x=model[1].x-(j-1)*x_interval;
        model[((i-1)*3)+j].y=model[1].y-(i-1)*y_interval;
              }
}

void COrder::OnPaint()
{

  CPaintDC dc(this); // device context for painting
  if(flagtimer==0){
          this->SetTimer(1,300,(TIMERPROC) NULL);   flagtimer=1;
            if(value_set_flag2==1) {
              this->GetDlgItem(IDC_EDIT1)->EnableWindow(false);
              this->GetDlgItem(IDC_EDIT2)->EnableWindow(false);
              this->GetDlgItem(IDC_EDIT3)->EnableWindow(false);
              UpdateData(false);}}
// Sleep(800);
  drawimage();
  // TODO: Add your message handler code here

  // Do not call CDialog::OnPaint() for painting messages
}

void COrder::drawimage()
{
  int i=0;
  CDC MemDC,MemDC2;
  CDC *pDC;
  CBitmap bt;
  CRect rc;
  GetClientRect(&rc);
  pDC = this->GetDC();
  bt.CreateCompatibleBitmap(pDC,rc.Width(),rc.Height());
  MemDC.CreateCompatibleDC(pDC);
  MemDC2.CreateCompatibleDC(pDC);

  MemDC.SelectObject(btbt);
  MemDC2.SelectObject(bt);
MemDC2.StretchBlt(myrect.left,myrect.top,myrect.Width(),myrect.Height(),&MemDC,0,0,w,h,SR
CCOPY);
```

```
CString temp(lpszString2);
temp+=" Camera Calibration Parameters";

CFont newFont;
CFont *pOldFont;
newFont.CreateFont(20,13,0,0,0,0,0,0,0,0,OUT_DEFAULT_PRECIS,CLIP_DEFAULT_PRECIS,
PROOF_QUALITY,FF_DONTCARE,"Arial");

pOldFont = pDC->SelectObject(&newFont);

pDC->SetTextColor(RGB(10,50,15));
pDC->SetBkMode(TRANSPARENT);

pDC->TextOut( 21, 30, temp);
pDC->SelectObject(pOldFont);

newFont.DeleteObject();

//MemDC2.FillSolidRect(&model_rect,RGB(100,155,255));

  CBrush newBrush;
  CBrush* oldBrush;
  newBrush.CreateSolidBrush(RGB(100,12,110));
  oldBrush=MemDC2.SelectObject(&newBrush);

for(i=1;i<=9;i++)
{
  if( rect_highlight[i]==1 &&  rect_highlight2[i]!=1)
  {MemDC2.Ellipse(&circle_rect[i]); break;}

}

MemDC2.SelectObject(oldBrush);
newBrush.DeleteObject();




  newBrush.CreateSolidBrush(RGB(0,0,255));
  oldBrush=MemDC2.SelectObject(&newBrush);

CString ctemp="";
int vtemp;

newFont.CreateFont(20,13,0,0,0,0,0,0,0,0,OUT_DEFAULT_PRECIS,CLIP_DEFAULT_PRECIS,
PROOF_QUALITY,FF_DONTCARE,"Arial");
      pOldFont = MemDC2.SelectObject(&newFont);
      MemDC2.SetTextColor(RGB(100,155,255));
MemDC2.SetBkMode(TRANSPARENT);

for(i=1;i<=9;i++)
{
```

```
 if( rect_highlight2[i]==1 )
    {
   vtemp=order[i]+48;
   ctemp=vtemp;


   MemDC2.Ellipse(&circle_rect[i]);
   MemDC2.DrawText(ctemp, &circle_rect[i], DT_CENTER   );
   MemDC2.TextOut(circle_rect[i].left,circle_rect[i].top,ctemp);

  }
}

MemDC2.SelectObject(oldBrush);
newBrush.DeleteObject();
int model_diameter=12;
newBrush.CreateSolidBrush(RGB(10,222,10));
oldBrush=pDC->SelectObject(&newBrush);

CFont newFont2;
CFont *pOldFont2;
  newFont2.CreateFont(20,13,0,0,0,0,0,0,0,OUT_DEFAULT_PRECIS,CLIP_DEFAULT_PREC
IS,PROOF_QUALITY,FF_DONTCARE,"Arial");

pOldFont2 = pDC->SelectObject(&newFont2);

pDC->SetTextColor(RGB(100,155,255));
pDC->SetBkMode(TRANSPARENT);
      int x_offset= -7 ;
int y_offset=  16;
//pDC->SetTextColor( RGB(100,255,100) );
for(i=1;i<=9;i++)
{
  vtemp=i+48;
    ctemp=vtemp;

  if( i!=count+1 ){
  pDC->Ellipse(model[i].x-model_diameter,model[i].y-
model_diameter,model[i].x+model_diameter,model[i].y+model_diameter);
  pDC->TextOut(model[i].x+x_offset,model[i].y+y_offset,ctemp);
  }
}

pDC->SelectObject(oldBrush);
newBrush.DeleteObject();

if(colorflag==1)  {newBrush.CreateSolidBrush(RGB(255,0,0)); pDC->SetTextColor(
RGB(255,0,0) );}
else  {newBrush.CreateSolidBrush(RGB(0,255,0));   }
  oldBrush=pDC->SelectObject(&newBrush);
```

```
i=count+1;
  vtemp=i+48;
    ctemp=vtemp;
pDC->TextOut(model[i].x+x_offset,model[i].y+y_offset,ctemp);
  pDC->Ellipse(model[i].x-model_diameter,model[i].y-
model_diameter,model[i].x+model_diameter,model[i].y+model_diameter);


pDC->SelectObject(oldBrush);
newBrush.DeleteObject();

pDC->SelectObject(pOldFont2);

newFont2.DeleteObject();



//MemDC2.StretchBlt(myrect.left,myrect.top,myrect.Width(),myrect.Height(),&MemDC,0,0,w,h,
SRCCOPY);



pDC->BitBlt(myrect.left,myrect.top, myrect.Width(),myrect.Height(),
&MemDC2,myrect.left,myrect.top, SRCCOPY );


if(count==9){

  KillTimer(1);
}

// for(i=1;i<=9;i++)
  // rect_highlight[i]=0;
//
}

void COrder::OnDont()
{
MessageBox("You may choose the order at a later time","Calibration");
  CDialog::OnCancel ();
  }
void COrder::OnOK()
{
  // TODO: Add extra validation here
  MPoint temp_pts[11];
  if( count!=9)
  { MessageBox("All circles must be marked","Calibration"); return;}

  int i,j;
  for(i=1;i<=9;i++){
```

```
    j=1;
    while(order[j]!=i) j++;
    temp_pts[i]=ptspts[j];
  }

  for(i=1;i<=9;i++)
  ptspts[i]=temp_pts[i];


  if(value_set_flag2==0){
  UpdateData(true);
  if(m_x==0 && m_y==0 && m_z==0) {MessageBox("Not all values can be
zero","Calibration"); return;}
  *X=(double)m_x;
  *Y=(double)m_y;
  *Z=(double)m_z;
  }
  CDialog::OnOK();
}

void COrder::OnLButtonDown(UINT nFlags, CPoint point)
{
  int i=0;

if(count==9) return;
double xx,yy,ll;

for(i=1;i<=9;i++)
{
  xx=point.x-temp_points[i].x;
  yy=point.y-temp_points[i].y;
  ll=(xx*xx)+(yy*yy);
  ll=sqrt(ll);
  if(rect_highlight2[i]!=1){
    if(ll<= 10){
    rect_highlight2[i]=1;
    count++;
    order[i]=count;

    }
  }
}
InvalidateRect( &myrect, false );

  CDialog::OnLButtonDown(nFlags, point);
}
void COrder::OnMouseMove(UINT nFlags, CPoint point)
{if(count==9) return;
  int i=0;

  int this_circle=0;
double xx,yy,ll;
```

```
 xx=point.x-prev_point.x;
 yy=point.y-prev_point.y;
 ll=(xx*xx)+(yy*yy);
 ll=sqrt(ll);

 if(ll==0) {prev_point=point; return;}
for(i=1;i<=9;i++)
{
 xx=point.x-temp_points[i].x;
 yy=point.y-temp_points[i].y;
 ll=(xx*xx)+(yy*yy);
 ll=sqrt(ll);

 if(ll<= 10)
 {rect_highlight[i]=1; this_circle=i;}

 else
   rect_highlight[i]=0;
}

if(this_circle!=prev_circle) InvalidateRect( &myrect, false );
prev_circle=this_circle;
prev_point=point;
  CDialog::OnMouseMove(nFlags, point);
}

void COrder::OnTimer(UINT nIDEvent)
{
if(nIDEvent!=1) return;
// Invalidate();
if(colorflag==1)  colorflag=0;
else colorflag=1;
CRect b(600,401,601,402);
InvalidateRect( &b, false );


  CDialog::OnTimer(nIDEvent);
}

void COrder::OnReset()
{
flagtimer=0;
    int i;
  for(i=1;i<=9;i++){
    rect_highlight[i]=0;
    rect_highlight2[i]=0;
  order[i]=0;
  }
count=0;
Invalidate();
}
```

```
void COrder::OnButton1()
{
exit(0);
}

////////////////////// order.h//////////////////////////////
// COrder dialog
// header file for the order class

#include "camera.h"
#if
!defined(AFX_ORDER_H__2AA259F9_7348_41A3_8900_3FC325D4333D__INCLUDED_)
#define AFX_ORDER_H__2AA259F9_7348_41A3_8900_3FC325D4333D__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
// Order.h : header file
//


class COrder : public CDialog
{
// Construction
public:
  void drawimage(void);
  void loadimage(MPoint *pts, CBitmap *bt,int width, int height,LPCTSTR lpszString,int
value_set_flag,double* x,double* y,double* z);
  COrder(CWnd* pParent = NULL);   // standard constructor
  CBitmap* btbt;
  CPoint model[11];
  MPoint* ptspts;
  int rect_highlight[11];
  int rect_highlight2[11];
  int w;
    int value_set_flag2;
  int h;
  int order[11];
  int completion_flag;
  int flagtimer;
  CPoint temp_points[11];
  CRect myrect;
  CRect circle_rect[11];
  LPCTSTR lpszString2;
  CPoint prev_point;
  int colorflag;
  CRect model_rect;
  int prev_circle;
  int count;
  double* X;
```

```
  double* Y;
  double* Z;

// Dialog Data
  //{{AFX_DATA(COrder)
  enum { IDD = IDD_DIALOG2 };
  double m_x;
  double m_y;
  double m_z;
  //}}AFX_DATA


// Overrides
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(COrder)
  protected:
  virtual void DoDataExchange(CDataExchange* pDX);   // DDX/DDV support
  //}}AFX_VIRTUAL

// Implementation
protected:

  // Generated message map functions
  //{{AFX_MSG(COrder)
  virtual void OnCancel();
  afx_msg void OnPaint();
  afx_msg void OnDont();
  virtual void OnOK();
  afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
  afx_msg void OnMouseMove(UINT nFlags, CPoint point);
  afx_msg void OnTimer(UINT nIDEvent);
  afx_msg void OnReset();
  afx_msg void OnButton1();
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the previous line.

#endif //
!defined(AFX_ORDER_H__2AA259F9_7348_41A3_8900_3FC325D4333D__INCLUDED_)
```

// robot.cpp: implementation of the robot class.
//routines for robot movement relted to calibration

```
//////////////////////////////////////////////////////////////////////
#include "stdafx.h"
#include "calibrb.h"
#include "robot.h"
#include "math.h"
#include <fstream>
```

```
#include <iostream>
#define rob_y 40.000
#define rob_x 60.000
#define pi 3.1415
#define sliding (54.8000+180)
#define horizon 88
#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif
using namespace std;
#define XX 49.95
#define YY 5.1

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

robot::robot()
{

}

robot::~robot()
{
}
void robot::initilize()
{BOOL success=true;
int car1[2]={0x4E,0},car2[2]={0x45,0},car3[2]={0x53,0},car4[2]={0x54,0},car5[2]={0x0D,0};
  DCB dcb;
  DWORD err;
  FillMemory(&dcb,sizeof(dcb),0);
  dcb.DCBlength=sizeof(dcb);

// COMMTIMEOUTS timeout;
  hComm = CreateFile("COM1", GENERIC_WRITE, 0, 0,
OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL , 0);
  if(hComm==INVALID_HANDLE_VALUE)
  {err=GetLastError();

  exit (0);}

  if((BuildCommDCB("COM1:2400,n,8,2",&dcb)<0))
  {
    exit(0);
  }
// success = GetCommState(hComm, &dcb);
err=GetLastError();
  if (!success) {err=GetLastError();}
err=GetLastError();
  dcb.fOutxCtsFlow=1;
```

```
  dcb.fOutxDsrFlow=1;
   dcb.BaudRate = CBR_2400;
  dcb.StopBits=2;
err=GetLastError();
//set the dcb values
   success = SetCommState(hComm, &dcb);
   if (!success)
   {err=GetLastError();

  exit (0);}
EscapeCommFunction(hComm, CLRDTR);
EscapeCommFunction(hComm, CLRRTS);
EscapeCommFunction(hComm, SETRTS);
EscapeCommFunction(hComm, SETDTR);
}
void robot::NEST()
{
SEND("RS");
SEND("NT");
SEND("SP9");
SEND("MI -6000,-2600,1800,1200,-1200,0");
SEND("MI 0,0,0,295,295,0");
SEND("HO");
SEND("GP 5,5,5");
}

void robot::SEND(char *tt)
{


  DWORD numWrite,numbit;
COMSTAT status;
  BOOL w,success=true;
  int carriage[2]={0x0D,0},bv[2]={0x07,0};
    int DONE=false;

    while(!DONE){
    w=ClearCommError(hComm, &numbit,&status);
    if(status.fCtsHold==false && status.fDsrHold==false )
      {

        if(*tt==NULL)
      {

          DONE=true;
          success= WriteFile(hComm, carriage, 1, &numWrite, 0);
      break;
      }
      if((WriteFile(hComm, tt, 1, &numWrite, 0)))
      {
        ++tt;
      }}
```

```
   }}
void robot::movetotext(char *fname)
{
fstream f;
f.open(fname);

CString string1="PS 1,";
CString string2=",";

int a[10];

int i;
char temp[10];
for(i=0;i<7;i++) f>>a[i];


for(i=0;i<6;i++){
itoa(a[i],temp,10);
string1+=temp;
if(i!=5) string1+=string2;
}
//if(AfxMessageBox(string1.GetBuffer(string1.GetLength()),MB_YESNO    )== IDOK)
{SEND(string1.GetBuffer(string1.GetLength()));
SEND("MO 1");}f.close();
}
void robot::moveto(double X, double Y, double Z, double A, double S)
{
  X=rob_x-(0+X);
  Y=rob_y-(Y-0);
double d5=21.46;


int i;
int a[10];
char temp[10];
double magnitude=sqrt((X*X)+(Y*Y));
double px,py,pz;
double A_an=A;
A=(double)(A*pi);A/=(double)180;


for(i=0;i<7;i++) a[i]=0;

//wrist vector
px=(double)(X-((double)d5*(X/magnitude)*sin(A)));
py=(double)(Y-((double)d5*(Y/magnitude)*sin(A)));
pz=(double)(Z-((double)d5*cos(A)));
pz-=25.0000;

double pmag=px*px+py*py+pz*pz;
pmag=sqrt(pmag);
```

```
//body rotation
double b;

b=(double)(atan2(Y,X));
b=fabs(b);
if(Y<0) b=b*-1;
a[0]=(int)(-40*b*180/pi);
double theta1=b*180/pi;
//if(a[0]> 5999 || a[0]<-5999) MessageBox("waist out of range");



//shoulder rotation
double lambda=0;
double beta=0;
double temp1=0;
temp1=pz/(sqrt((px*px)+(py*py)));
lambda=atan(temp1);

temp1=px*px+py*py+pz*pz+22.000*22.000-16.00*16.00;
temp1/=(2*22.000*pmag);
beta=acos(temp1);

b=(double)(beta+lambda)*180.00;b/=pi;b=b-(double)35.0000;
b=b*40.000;

a[1]=(int)b;

//elbow rotation

double alpha=0;
alpha=asin((22.00/16.00)*sin(beta));
b=alpha+beta;
b=b*-40.00*180.00;b/=pi;
b=b+1800.00;
a[2]=(int)b;

//wrist angles

double theta4,theta5;

theta4=A_an-90.00+(((lambda-alpha)*180.00)/pi);
theta4+=1.9;

theta5=(S-theta1);


a[3]=(int)((1200+0)-((double)(40.000/3.000)*(theta4+theta5)));
a[4]=(int)((1200+0)+((double)(40.000/3.000)*(theta4-theta5)));

a[5]=0;
CString string1="PS 1,";
```

```
CString string2=",";

for(i=0;i<6;i++){
itoa(a[i],temp,10);
string1+=temp;
if(i!=5) string1+=string2;
}

//if(AfxMessageBox(string1.GetBuffer(string1.GetLength()),MB_YESNO    )== IDYES )
{
SEND(string1.GetBuffer(string1.GetLength()));

SEND("MO 1");

}}
void robot::OPEN()
{
SEND("GO");
}

void robot::CLOSE()
{
SEND("GC");
}

void robot::cartesian_text(char *p)
{
}
void robot::HOME()
{
SEND("OG");
}

void robot::calibrb(int a)
{
  if(a==2){

moveto(25, 40,5, 180,horizon);
  }

    if(a==3){

moveto(25, 40,9, 180,horizon);
  }


      if(a==4){
moveto(25, 40,13, 180,horizon);
  }
      if(a==1){

moveto(25, 40,-0.7, 180,horizon);
```

```
 }}
void robot::putcalibback()
{

SEND("GF 1");
moveto(XX, YY,-.5, 180,sliding);
SEND("GO");
SEND("GF 0");
moveto(XX, YY,2, 180,sliding);

//SEND("OG");
}
void robot::MI(int v1, int v2, int v3, int v4, int v5)
{
CString string1="MI ";
CString string2=",";

int a[10];
a[1]=v1 ;
a[2]=v2 ;
a[3]=v3 ;
a[4]=v4 ;
a[5]=v5 ;

int i;
char temp[10];


for(i=1;i<6;i++){
itoa(a[i],temp,10);
string1+=temp;
string1+=string2;
}
string1+="0";



SEND(string1.GetBuffer(string1.GetLength()));


}

void robot::reset()
{
SEND("RS");
}

void robot::grabtarget()
{
moveto(XX, YY,2, 180,sliding);
moveto(XX, YY,-.9, 180,sliding);
CLOSE();
```

```
SEND("GF 1");
}

void robot::closecomm()
{
CloseHandle(hComm);
}

void robot::home2()
{
moveto(XX, YY,8, 180,sliding);
}
```

// robot.h: interface for the robot class.

```
//
//////////////////////////////////////////////////////////////////////

#if
!defined(AFX_ROBOT_H__231B518D_AE22_4164_85F0_75307EF40B6A__INCLUDED_)
#define AFX_ROBOT_H__231B518D_AE22_4164_85F0_75307EF40B6A__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

class robot
{
public:
  void home2(void);
  void closecomm(void);
  void grabtarget(void);
  void reset(void);
  void MI(int v1, int v2, int v3, int v4, int v5);
  void putcalibback(void);
  void calibrb(int a);
  void HOME(void);
  void cartesian_text(char* p);
  void CLOSE(void);
  void OPEN(void);
  void movetotext(char* fname);
  void moveto(double X, double Y, double Z, double A, double S);
  HICON m_hIcon;
  HANDLE hComm;
  void SEND(char* t);
  void NEST(void);
  void initilize(void);
  robot();
  virtual ~robot();

};

#endif //
```

```
!defined(AFX_ROBOT_H__231B518D_AE22_4164_85F0_75307EF40B6A__INCLUDED_)
```

// calibrbView.cpp : implementation of the CcalibrbView class
//the bulk of the calibration routeines is here (except for the Singular
//Value Decomposition routine which has purposefully been left out, see
//"numerical Recipies in C" for code

```
#include "stdafx.h"
#include "robot.h"
#include "calibrb.h"
#include "calibrbDoc.h"
#include "calibrbView.h"
#include "ORRESPOND.h"
#include <process.h>

#define num_points 9
#define num_planes 3
#define plane_num 3
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
//
// CcalibrbView

IMPLEMENT_DYNCREATE(CcalibrbView, CScrollView)

BEGIN_MESSAGE_MAP(CcalibrbView, CScrollView)
  //{{AFX_MSG_MAP(CcalibrbView)
  ON_COMMAND(ID_CAMERA_GRAB, OnCameraGrab)
  ON_COMMAND(ID_CAMERA_CALIBRATE, OnCameraCalibrate)
  ON_COMMAND(ID_SHOW_LEFTIMAGE, OnShowLeftimage)
  ON_COMMAND(ID_SHOW_RIGHTIMAGE, OnShowRightimage)
  ON_COMMAND(ID_SHOW_TLEFTIMAGE, OnShowTleftimage)
  ON_COMMAND(ID_SHOW_TRIGHTIMAGE, OnShowTrightimage)
  ON_COMMAND(ID_SHOW_LEFTBORDER, OnShowLeftborder)
  ON_COMMAND(ID_SHOW_RIGHTBORDER, OnShowRightborder)
  ON_COMMAND(ID_CAMERA_RELEASEANDCLOSE, OnCameraReleaseandclose)
  ON_COMMAND(ID_SHOW_SHOWCENTRES, OnShowShowcentres)
  ON_COMMAND(ID_CAMERA_SHOWCALIBSTEPS, OnCameraShowcalibsteps)
  ON_COMMAND(ID_LEVEL_1, OnLevel1)
  ON_COMMAND(ID_LEVEL_2, OnLevel2)
  ON_COMMAND(ID_LEVEL_3, OnLevel3)
  ON_COMMAND(ID_CALIBRATE_NEST, OnCalibrateNest)
  ON_COMMAND(ID_CALIBRATE_GRAB, OnCalibrateCalibrate)
  ON_COMMAND(ID_ROBOT_HOME, OnRobotHome)
  ON_COMMAND(ID_ROBOT_OPEN, OnRobotOpen)
  ON_COMMAND(ID_ROBOT_PUTBACK, OnRobotPutback)
```

```
  ON_COMMAND(ID_DEBUG_WRITEPOINTS, OnDebugWritepoints)
  ON_COMMAND(ID_CALIBRATE_START, OnCalibrateStart)
  ON_COMMAND(ID_ROBOT_RESET, OnRobotReset)
  ON_COMMAND(ID_ROBOT_MOVEASIDE, OnRobotMoveaside)
  //}}AFX_MSG_MAP
  // Standard printing commands
  ON_COMMAND(ID_FILE_PRINT, CScrollView::OnFilePrint)
  ON_COMMAND(ID_FILE_PRINT_DIRECT, CScrollView::OnFilePrint)
  ON_COMMAND(ID_FILE_PRINT_PREVIEW, CScrollView::OnFilePrintPreview)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
//
// CcalibrbView construction/destruction

CcalibrbView::CcalibrbView()
{
  // TODO: add construction code here
flag1=0;
flag3=0;
flag_centres=0;
calibrate_flag=0;
int i;


cam_calibL=new double [14];
cam_calibR=new double [14];


for(i=0;i<=12;i++){
cam_calibL[i]=0;
cam_calibR[i]=0;}

FR= new double* [6];
FL= new double* [6];
MR= new double* [6];
ML= new double* [6];

for (i=0; i<6; i++)
{FR[i] = new double[5];
FL[i] = new double[5];
MR[i] = new double[5];
ML[i] = new double[5];
}


}

CcalibrbView::~CcalibrbView()
{
}
```

```
BOOL CcalibrbView::PreCreateWindow(CREATESTRUCT& cs)
{
  // TODO: Modify the Window class or styles here by modifying
  //  the CREATESTRUCT cs

  return CScrollView::PreCreateWindow(cs);
}

////////////////////////////////////////////////////////////////////////////
//
// CcalibrbView drawing

void CcalibrbView::OnDraw(CDC* pDC)
{
  CcalibrbDoc* pDoc = GetDocument();
  ASSERT_VALID(pDoc);

  if(flag_centres==1){
    writecentres(pDC);
  }else
  if(flag1==1){
CDC memdc;
memdc.CreateCompatibleDC(pDC);
CBitmap* pOldBm = memdc.SelectObject(curb);
CRect rc;
GetClientRect(&rc);
pDC->BitBlt(0, 0, 700,700, &memdc, 0, 0, SRCCOPY );
memdc.SelectObject(pOldBm);
DeleteDC(memdc);
}

  // TODO: add draw code for native data here
}

void CcalibrbView::OnInitialUpdate()
{
  CScrollView::OnInitialUpdate();

  CSize sizeTotal;
  // TODO: calculate the total size of this view
  sizeTotal.cx = sizeTotal.cy = 1000;
  SetScrollSizes(MM_TEXT, sizeTotal);
AfxGetMainWnd()->SetWindowText ( "Stereo Project" );
//changing the title
level=1;

}

////////////////////////////////////////////////////////////////////////////
//
// CcalibrbView printing
```

```
BOOL CcalibrbView::OnPreparePrinting(CPrintInfo* pInfo)
{
  // default preparation
  return DoPreparePrinting(pInfo);
}

void CcalibrbView::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
  // TODO: add extra initialization before printing

}

void CcalibrbView::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
  // TODO: add cleanup after printing
}

/////////////////////////////////////////////////////////////////////////////
//
// CcalibrbView diagnostics

#ifdef _DEBUG
void CcalibrbView::AssertValid() const
{
  CScrollView::AssertValid();
}

void CcalibrbView::Dump(CDumpContext& dc) const
{
  CScrollView::Dump(dc);
}

CcalibrbDoc* CcalibrbView::GetDocument() // non-debug version is inline
{
  ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CcalibrbDoc)));
  return (CcalibrbDoc*)m_pDocument;
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
//
// CcalibrbView message handlers


void CcalibrbView::OnCameraCalibrate()
{int i,j;
  double X=0,Y=0,Z=0;
int cc=0;
```

```
calibrate_flag=1;//set the flag
//fix orders
  for(i=1;i<=3;i++){
MyCam.getworldpos3(&X,&Y,&Z,i);
COrder centroid_dialog1;
centroid_dialog1.loadimage(MyCentres[i].RPoints,&S_IMAGE[i].RImage,MyCam.S_X,MyCam.S
_Y,"Right",1,&X,&Y,&Z);
centroid_dialog1.DoModal ();
COrder centroid_dialog2;
centroid_dialog2.loadimage(MyCentres[i].LPoints,&S_IMAGE[i].LImage,MyCam.S_X,MyCam.S_
Y,"Left",1,&X,&Y,&Z);
centroid_dialog2.DoModal ();

centroid_dialog1.DestroyWindow( );
centroid_dialog2.DestroyWindow( );
MyCam.getworldpos2(MyCentres[i].WPoints,X,Y,Z);

}

//declare normalization matrix now

double** norm_imageL;
double** norm_imageR;
double** norm_space;

double** left_calib;
double** right_calib;

double** A;
double** v;



double* b=new double[1+(2*num_points*num_planes)];
double* w=new double[1+(2*num_points*num_planes)];
double* C_left=new double[5];
double* C_right=new double[5];

A= new double* [1+(2*num_points*num_planes)];
v= new double* [1+(2*num_points*num_planes)];

 left_calib=new double* [6];
 right_calib=new double* [6];
 norm_imageL=new double* [6];
 norm_imageR=new double* [6];
 norm_space=new double* [6];
```

```
for (i=0; i<1+(2*num_points*num_planes); i++)
{ A[i] = new double[13];
 v[i] = new double[13];
 }

 for (i=0; i<6; i++)
{
  left_calib[i] = new double[6];
  right_calib[i] = new double[6];
  norm_imageL[i] = new double[6];
  norm_imageR[i] = new double[6];
  norm_space[i] = new double[6];
 }
//ninitliaze to zero
for(j=0;j<=5;j++)
for(i=0;i<=5;i++){
norm_imageL[j][i]=0;
norm_imageR[j][i]=0;
norm_space[j][i]=0;
left_calib[j][i]=0;
right_calib[j][i]=0;
ML[j][i]=0;
MR[j][i]=0;
}


//put real values inside another array to keep unnormalized values for later

for (j=1; j<=num_planes; j++)
for (i=1; i<=num_points; i++)
  {
MyCentres_real[j].WPoints[i].x=MyCentres[j].WPoints[i].x;
MyCentres_real[j].WPoints[i].y=MyCentres[j].WPoints[i].y;
MyCentres_real[j].WPoints[i].z=MyCentres[j].WPoints[i].z;

MyCentres_real[j].LPoints[i].x=MyCentres[j].LPoints[i].x;
MyCentres_real[j].LPoints[i].y=MyCentres[j].LPoints[i].y;

MyCentres_real[j].RPoints[i].y=MyCentres[j].RPoints[i].y;
MyCentres_real[j].RPoints[i].x=MyCentres[j].RPoints[i].x;
}
//now normalize

MyCam.normalize_image( norm_imageL,norm_imageR,norm_space ,MyCentres);

for (i=0; i<=(2*num_points*num_planes); i++) {b[i]=0; w[i]=0;}


for (i=1; i<=(2*num_points*num_planes); i++)
  for (j=1; j<=12; j++)
  {
  A[i][j]=0;
```

```
v[i][j]=0;
}


int ccc=1;
int k=0;

for (j=1; j<=num_planes; j++)
for (i=1; i<=num_points; i++)
 {
if(num_planes==1) j=plane_num;


for(k=1;k<=4;k++) A[(ccc*2)-1][k]=0;



A[(ccc*2)-1][5]=-MyCentres[j].WPoints[i].x;
A[(ccc*2)-1][6]=-MyCentres[j].WPoints[i].y;
A[(ccc*2)-1][7]=-MyCentres[j].WPoints[i].z;
A[(ccc*2)-1][8]=-1;


A[(ccc*2)-1][9]=MyCentres[j].WPoints[i].x*MyCentres[j].LPoints[i].y;
A[(ccc*2)-1][10]=MyCentres[j].WPoints[i].y*MyCentres[j].LPoints[i].y;
A[(ccc*2)-1][11]=MyCentres[j].WPoints[i].z*MyCentres[j].LPoints[i].y;
A[(ccc*2)-1][12]=1*MyCentres[j].LPoints[i].y;


//next line


A[ccc*2][1]=MyCentres[j].WPoints[i].x;
A[ccc*2][2]=MyCentres[j].WPoints[i].y;
A[ccc*2][3]=MyCentres[j].WPoints[i].z;
A[ccc*2][4]=1;



for(k=5;k<=8;k++) A[ccc*2][k]=0;







A[ccc*2][9]=MyCentres[j].WPoints[i].x*-1*MyCentres[j].LPoints[i].x;
A[ccc*2][10]=MyCentres[j].WPoints[i].y*-1*MyCentres[j].LPoints[i].x;
A[ccc*2][11]=MyCentres[j].WPoints[i].z*-1*MyCentres[j].LPoints[i].x;
A[ccc*2][12]=1*-1*MyCentres[j].LPoints[i].x;

ccc++;
```

```
}

/////////
FILE* tempf;
tempf=fopen("LmatrixA.txt","wt");
  fprintf(tempf,"A=[ ");
  for(i=1;i<=(2*num_points*num_planes);i++){
    if(i!=1) fprintf(tempf," \n ");
    for(j=1;j<=12;j++){
      fprintf(tempf,"%f ",A[i][j]);
  }}

fprintf(tempf," ] ");

fprintf(tempf,"\n\n\n ");
  fprintf(tempf,"b=[ ");

    for(j=1;j<=(2*num_points*num_planes);j++){
        if(j!=1) fprintf(tempf," ; ");
      fprintf(tempf,"%f ",b[j]);
            }

fprintf(tempf," ] ");
fclose(tempf);

MyCam.solve2(A,(2*num_points*num_planes),12,cam_calibL);
make2D(cam_calibL,left_calib);
cc=0;
tempf=fopen("Lcamera.txt","wt");
  fprintf(tempf,"P=[ ");
  for(i=1;i<4;i++){
    if(i!=1) fprintf(tempf," ;");
    for(j=1;j<5;j++){
      cc++;
    fprintf(tempf,"%f ",cam_calibL[cc]);
  }}

fprintf(tempf," ] ");

fprintf(tempf," \n\n\n\n");
  fprintf(tempf,"v=[ ");
  for(i=1;i<=12;i++){
    if(i!=1) fprintf(tempf," \n ");
    for(j=1;j<=12;j++){
      fprintf(tempf,"%f ",v[i][j]);
  }}

fprintf(tempf," ] ");

  fprintf(tempf," \n\n\n\n");
  fprintf(tempf,"w=[ ");
  for(i=1;i<=12;i++){
```

```
   fprintf(tempf,"%f ",w[i]);
  }fprintf(tempf," ] ");


fclose(tempf);


//////////////////////////////////////////////////////
///right camera//////////////////////////////////////
for (i=0; i<=(2*num_points*num_planes); i++) {b[i]=0; w[i]=0;}
for (i=1; i<=(2*num_points*num_planes); i++)
  for (j=1; j<=12; j++)
  {
  A[i][j]=0;
  v[i][j]=0;
  }
ccc=1;

for (j=1; j<=num_planes; j++)
for (i=1; i<=num_points; i++)
  {
if(num_planes==1) j=plane_num;


for(k=1;k<=4;k++) A[(ccc*2)-1][k]=0;



  A[(ccc*2)-1][5]=-MyCentres[j].WPoints[i].x;
  A[(ccc*2)-1][6]=-MyCentres[j].WPoints[i].y;
  A[(ccc*2)-1][7]=-MyCentres[j].WPoints[i].z;
  A[(ccc*2)-1][8]=-1;

  A[(ccc*2)-1][9]=MyCentres[j].WPoints[i].x*MyCentres[j].RPoints[i].y;
  A[(ccc*2)-1][10]=MyCentres[j].WPoints[i].y*MyCentres[j].RPoints[i].y;
  A[(ccc*2)-1][11]=MyCentres[j].WPoints[i].z*MyCentres[j].RPoints[i].y;
  A[(ccc*2)-1][12]=1*MyCentres[j].RPoints[i].y;


  //next line


  A[ccc*2][1]=MyCentres[j].WPoints[i].x;
  A[ccc*2][2]=MyCentres[j].WPoints[i].y;
  A[ccc*2][3]=MyCentres[j].WPoints[i].z;
  A[ccc*2][4]=1;



  for(k=5;k<=8;k++) A[ccc*2][k]=0;
```

```
  A[ccc*2][9]=MyCentres[j].WPoints[i].x*-1*MyCentres[j].RPoints[i].x;
  A[ccc*2][10]=MyCentres[j].WPoints[i].y*-1*MyCentres[j].RPoints[i].x;
  A[ccc*2][11]=MyCentres[j].WPoints[i].z*-1*MyCentres[j].RPoints[i].x;
  A[ccc*2][12]=1*-1*MyCentres[j].RPoints[i].x;

  ccc++;
}

/////////

tempf=fopen("RmatrixA.txt","wt");
  fprintf(tempf,"A=[ ");
  for(i=1;i<=(2*num_points*num_planes);i++){
    if(i!=1) fprintf(tempf," \n ");
    for(j=1;j<=12;j++){
      fprintf(tempf,"%f ",A[i][j]);
  }}

fprintf(tempf," ] ");

fprintf(tempf,"\n\n\n ");
  fprintf(tempf,"b=[ ");

    for(j=1;j<=(2*num_points*num_planes);j++){
        if(j!=1) fprintf(tempf," ; ");
      fprintf(tempf,"%f ",b[j]);
            }

fprintf(tempf," ] ");
fclose(tempf);
MyCam.solve2(A,(2*num_points*num_planes),12,cam_calibR);
make2D(cam_calibR,right_calib);
cc=0;
tempf=fopen("Rcamera.txt","wt");
  fprintf(tempf,"P=[ ");
  for(i=1;i<4;i++){
    if(i!=1) fprintf(tempf," ;");
    for(j=1;j<5;j++){
      cc++;
    fprintf(tempf,"%f ",cam_calibR[cc]);
  }}

fprintf(tempf," ] ");

fprintf(tempf," \n\n\n\n");
  fprintf(tempf,"v=[ ");
  for(i=1;i<=12;i++){
    if(i!=1) fprintf(tempf," \n ");
    for(j=1;j<=12;j++){
      fprintf(tempf,"%f ",v[i][j]);
  }}
```

```
fprintf(tempf," ] ");

  fprintf(tempf," \n\n\n\n");
  fprintf(tempf,"w=[ ");
  for(i=1;i<=12;i++){
    fprintf(tempf,"%f ",w[i]);
  }
fprintf(tempf," ] ");
fclose(tempf);
//if you wish to implement the non linear least square minimization
//using the levenberg-marquardt method read this:
//1- the minimization has to take place before DENORMALIZING
//2- the value of the projection matrices is in two two dimensional array of type double
//left_calib and right_calib
//note the denormalization is done in the next line, so the minimization has to take place before
that



//denormalize the parameters
MyCam.denormalize(norm_imageL,norm_imageR,left_calib,right_calib,norm_space);

//print right camera calibration parameters
tempf=fopen("rcalib.txt","wt");
// fprintf(tempf,"PR=[ ");
  for(i=1;i<4;i++){
  // if(i!=1) fprintf(tempf," ;");
    for(j=1;j<5;j++){
      fprintf(tempf,"%f ",right_calib[i][j]);
  }}
fprintf(tempf," ] ");
fclose(tempf);
//print left camera calibration parameters
tempf=fopen("Lcalib.txt","wt");
// fprintf(tempf," ");
  for(i=1;i<4;i++){
  // if(i!=1) fprintf(tempf,"   ");
    for(j=1;j<5;j++){
      fprintf(tempf,"%f  ",left_calib[i][j]);
  }}
fprintf(tempf," ");
fclose(tempf);

//put back the real values in the normalized coordinates
for (j=1; j<=num_planes; j++)
for (i=1; i<=num_points; i++)
  {
MyCentres[j].WPoints[i].x=MyCentres_real[j].WPoints[i].x;
MyCentres[j].WPoints[i].y=MyCentres_real[j].WPoints[i].y;
MyCentres[j].WPoints[i].z=MyCentres_real[j].WPoints[i].z;
```

```
MyCentres[j].LPoints[i].x=MyCentres_real[j].LPoints[i].x;
MyCentres[j].LPoints[i].y=MyCentres_real[j].LPoints[i].y;

MyCentres[j].RPoints[i].y=MyCentres_real[j].RPoints[i].y;
MyCentres[j].RPoints[i].x=MyCentres_real[j].RPoints[i].x;
}

//find mean square error
MyCam.finderror(&left_x_error,&left_y_error,&right_x_error,&right_y_error,left_calib,right_cali
b,MyCentres);


//find ccenteres of the cameras
MyCam.find_center(left_calib,C_left);
MyCam.find_center(right_calib,C_right);

//find the fundamental matrix

MyCam.findF(FL,FR,MyCentres);

//put left calib and right calin arrays into permanent public arrays
for(i=1;i<=3;i++)
for(j=1;j<=4;j++){
ML[i][j]=left_calib[i][j];
MR[i][j]=right_calib[i][j];
}


//free memory
for (i=0; i<1+(2*num_points*num_planes); i++)
{
  delete [] A[i] ;
  delete [] v[i] ;
}


for (i=0; i<6; i++)
{
  delete [] right_calib[i]   ;
  delete [] norm_imageL[i]   ;
  delete [] norm_imageR[i]   ;
  delete [] norm_space[i]    ;
  delete [] left_calib[i]    ;
 }


delete [] b;
delete [] w;
delete [] A;
delete [] v;
delete [] C_left;
```

```
delete [] C_right;

delete [] left_calib;
delete [] right_calib;
delete [] norm_imageL;
delete [] norm_imageR;
delete [] norm_space;

//////////////////////////
if(flag1==1){
  for(i=1;i<=3;i++){
S_IMAGE[i].RImage.DeleteObject( );
S_IMAGE[i].LImage.DeleteObject( );
S_IMAGE[i].TRImage.DeleteObject( );
S_IMAGE[i].TLImage.DeleteObject( );
S_IMAGE[i].CRImage.DeleteObject( );
S_IMAGE[i].CLImage.DeleteObject( );
}
}
flag1=0;//free the buffers
flag3=1; //indicate calibration is fiinshed



system("copy \"C:\\Documents and Settings\\Sid-Ahmed
robot\\Desktop\\calibrb\\left_f.txt\" \"C:\\Documents and Settings\\Sid-Ahmed
robot\\Desktop\\robsecond\\configs\\LF.txt\"");
system("copy \"C:\\Documents and Settings\\Sid-Ahmed
robot\\Desktop\\calibrb\\right_f.txt\" \"C:\\Documents and Settings\\Sid-Ahmed
robot\\Desktop\\robsecond\\configs\\RF.txt\"");
system("copy \"C:\\Documents and Settings\\Sid-Ahmed
robot\\Desktop\\calibrb\\Lcalib.txt\" \"C:\\Documents and Settings\\Sid-Ahmed
robot\\Desktop\\robsecond\\configs\\LM.txt\"");
system("copy \"C:\\Documents and Settings\\Sid-Ahmed
robot\\Desktop\\calibrb\\rcalib.txt\" \"C:\\Documents and Settings\\Sid-Ahmed
robot\\Desktop\\robsecond\\configs\\RM.txt\"");

}

void CcalibrbView::OnShowLeftimage()
{
  if(calibrate_flag==0) return;
  curb=&S_IMAGE[level].LImage;
flag_centres=0;
    Invalidate();

}

void CcalibrbView::OnShowRightimage()
{if(calibrate_flag==0) return;
  curb=&S_IMAGE[level].RImage;
  flag_centres=0;
```

```
    Invalidate();

}

void CcalibrbView::OnShowTleftimage()
{if(calibrate_flag==0) return;
  curb=&S_IMAGE[level].TLImage;
  flag_centres=0;
    Invalidate();

}

void CcalibrbView::OnShowTrightimage()
{if(calibrate_flag==0) return;
  curb=&S_IMAGE[level].TRImage;
    flag_centres=0;
    Invalidate();}

void CcalibrbView::OnShowLeftborder()
{if(calibrate_flag==0) return;
  curb=&S_IMAGE[level].CLImage;
flag_centres=0;
    Invalidate();
}


void CcalibrbView::OnShowRightborder()
{if(calibrate_flag==0) return;
  curb=&S_IMAGE[level].CRImage;
  flag_centres=0;
    Invalidate();

}



void CcalibrbView::OnCameraReleaseandclose()
{int i=0;
  if(flag1==1){
    for(i=1;i<=3;i++){
 S_IMAGE[i].RImage.DeleteObject( );
 S_IMAGE[i].LImage.DeleteObject( );}
}

  exit(0);
}

void CcalibrbView::OnShowShowcentres()
{if(calibrate_flag==0) return;
  flag_centres=1;
  Invalidate();
```

```
}

void CcalibrbView::writecentres(CDC *pDC)
{if(calibrate_flag==0) return;
int q=level;
int i;
CString stuff("");

char* temp=new char[390];
stuff+="          Left                                  Right
World\n";
stuff+="  ----------------------                ------------------------                    ------
----------------------------\n";
stuff+="    X            Y                        X           Y                        X
Y          Z  \n";
stuff+="---------------------------------          ---------------------------------          ----
-----------------------------------\n";
for(i=1;i<=9;i++)
{
//print the centes
  sprintf(temp," %0.1f        %0.1f                  %0.1f        %0.1f
%0.1f        %0.1f        %0.1f
\n",MyCentres[q].LPoints[i].x,MyCentres[q].LPoints[i].y,MyCentres[q].RPoints[i].x,MyCentres[q].
RPoints[i].y,MyCentres[q].WPoints[i].x,MyCentres[q].WPoints[i].y,MyCentres[q].WPoints[i].z);
  stuff+=temp;
}

//now print the mean errors
sprintf(temp,"\n\nMean Square Error in Left Camera X coordinate is: %0.5f\n",left_x_error);
stuff+=temp;
sprintf(temp,"Mean Square Error in Left Camera Y coordinate is: %0.5f\n",left_y_error);
stuff+=temp;
sprintf(temp,"Mean Square Error in Right Camera X coordinate is: %0.5f\n",right_x_error);
stuff+=temp;
sprintf(temp,"Mean Square Error in Right Camera Y coordinate is: %0.5f\n",right_y_error);
stuff+=temp;



CRect rc;
GetClientRect(rc);


rc.OffsetRect(5,5);
pDC->DrawText(stuff, -1, rc,DT_LEFT);


delete [] temp;


}
```

```
void CcalibrbView::OnCameraShowcalibsteps()
{
  if(calibrate_flag==0) return;
calibrate.Create(IDD_DIALOG1,this);
calibrate.ShowWindow(SW_SHOW);
calibrate.loadpic1(&S_IMAGE[level].RImage,&S_IMAGE[level].LImage,&S_IMAGE[level].TRI
mage,&S_IMAGE[level].TLImage,&S_IMAGE[level].CRImage,&S_IMAGE[level].CLImage,My
Cam.S_X,MyCam.S_Y);

}

void CcalibrbView::OnLevel1()
{
  level=1;Invalidate();

}
void CcalibrbView::OnLevel2()
{
  level=2;Invalidate();

}

void CcalibrbView::OnLevel3()
{
  level=3;Invalidate();

}

void CcalibrbView::OnCalibrateNest()
{
robot myrob;
myrob.initilize();
myrob.NEST();
myrob.closecomm();
}
void CcalibrbView::OnCalibrateCalibrate()
{
  int i=0;

if(flag1==1){
  for(i=1;i<=3;i++){
S_IMAGE[i].RImage.DeleteObject();
S_IMAGE[i].LImage.DeleteObject();
S_IMAGE[i].TRImage.DeleteObject();
S_IMAGE[i].TLImage.DeleteObject();
S_IMAGE[i].CRImage.DeleteObject();
S_IMAGE[i].CLImage.DeleteObject();
}
}
camera mycam;
robot myrob;
```

```
mycam.initialize();
myrob.initilize();
myrob.grabtarget();

myrob.calibrb(4);
myrob.calibrb(1);
//MessageBox("Continue");
Sleep(2000);
mycam.GRAB();

i=1;
S_IMAGE[i].width=mycam.S_X;
S_IMAGE[i].height=mycam.S_Y;
S_IMAGE[i].RImage.Attach(mycam.ArrayToBitmap(mycam.userImage1));
S_IMAGE[i].LImage.Attach(mycam.ArrayToBitmap(mycam.userImage2));
S_IMAGE[i].TRImage.Attach(mycam.ArrayToBitmap(mycam.userImage1));
S_IMAGE[i].TLImage.Attach(mycam.ArrayToBitmap(mycam.userImage2));
S_IMAGE[i].CRImage.Attach(mycam.ArrayToBitmap(mycam.userImage1));
S_IMAGE[i].CLImage.Attach(mycam.ArrayToBitmap(mycam.userImage2));
mycam.export();

mycam.threshold1(&S_IMAGE[i].TRImage);
mycam.threshold1(&S_IMAGE[i].TLImage);

mycam.border(MyCentres[i].RPoints,&S_IMAGE[i].TRImage,&S_IMAGE[i].CRImage);
mycam.border(MyCentres[i].LPoints,&S_IMAGE[i].TLImage,&S_IMAGE[i].CLImage);



myrob.calibrb(2);

//MessageBox("Continue");
Sleep(2000);
mycam.GRAB();

i=2;
S_IMAGE[i].width=mycam.S_X;
S_IMAGE[i].height=mycam.S_Y;
S_IMAGE[i].RImage.Attach(mycam.ArrayToBitmap(mycam.userImage1));
S_IMAGE[i].LImage.Attach(mycam.ArrayToBitmap(mycam.userImage2));
S_IMAGE[i].TRImage.Attach(mycam.ArrayToBitmap(mycam.userImage1));
S_IMAGE[i].TLImage.Attach(mycam.ArrayToBitmap(mycam.userImage2));
S_IMAGE[i].CRImage.Attach(mycam.ArrayToBitmap(mycam.userImage1));
S_IMAGE[i].CLImage.Attach(mycam.ArrayToBitmap(mycam.userImage2));
mycam.export();

mycam.threshold1(&S_IMAGE[i].TRImage);
mycam.threshold1(&S_IMAGE[i].TLImage);

mycam.border(MyCentres[i].RPoints,&S_IMAGE[i].TRImage,&S_IMAGE[i].CRImage);
mycam.border(MyCentres[i].LPoints,&S_IMAGE[i].TLImage,&S_IMAGE[i].CLImage);
myrob.calibrb(3);
```

```
//MessageBox("Continue");
Sleep(2000);
mycam.GRAB();

i=3;
S_IMAGE[i].width=mycam.S_X;
S_IMAGE[i].height=mycam.S_Y;
S_IMAGE[i].RImage.Attach(mycam.ArrayToBitmap(mycam.userImage1));
S_IMAGE[i].LImage.Attach(mycam.ArrayToBitmap(mycam.userImage2));
S_IMAGE[i].TRImage.Attach(mycam.ArrayToBitmap(mycam.userImage1));
S_IMAGE[i].TLImage.Attach(mycam.ArrayToBitmap(mycam.userImage2));
S_IMAGE[i].CRImage.Attach(mycam.ArrayToBitmap(mycam.userImage1));
S_IMAGE[i].CLImage.Attach(mycam.ArrayToBitmap(mycam.userImage2));
mycam.export();

mycam.threshold1(&S_IMAGE[i].TRImage);
mycam.threshold1(&S_IMAGE[i].TLImage);

mycam.border(MyCentres[i].RPoints,&S_IMAGE[i].TRImage,&S_IMAGE[i].CRImage);
mycam.border(MyCentres[i].LPoints,&S_IMAGE[i].TLImage,&S_IMAGE[i].CLImage);

//////////////////
MyCam.S_X=mycam.S_X;
MyCam.S_Y=mycam.S_Y;
//////////////////


curb=&S_IMAGE[3].RImage;
level=3;
flag1=1;
Invalidate();
calibrate_flag=1;


myrob.putcalibback();
myrob.home2();
myrob.closecomm();
mycam.release();
}

void CcalibrbView::OnRobotHome()
{robot myrob;
myrob.initilize();
  myrob.HOME();
myrob.closecomm();

}

void CcalibrbView::OnRobotOpen()
{
  robot myrob;
```

```
myrob.initilize();
myrob.CLOSE();
myrob.closecomm();

}

void CcalibrbView::OnRobotPutback()
{robot myrob;
myrob.initilize();
myrob.putcalibback();
  myrob.closecomm();

}

void CcalibrbView::make2D(double a[], double **b)
{
b[1][1]=a[1];
b[1][2]=a[2];
b[1][3]=a[3];
b[1][4]=a[4];
b[2][1]=a[5];
b[2][2]=a[6];
b[2][3]=a[7];
b[2][4]=a[8];
b[3][1]=a[9];
b[3][2]=a[10];
b[3][3]=a[11];
b[3][4]=a[12];
}

void CcalibrbView::OnDebugWritepoints()
{
  int i,j;

FILE *rightf2D,*leftf2D,*f3D;


rightf2D=fopen("Right2D.txt","wt");
f3D=fopen("3D.txt","wt");
leftf2D=fopen("Left2D.txt","wt");



fprintf(rightf2D,"27\n");
fprintf(leftf2D,"27\n");
fprintf(f3D,"27\n");

for(i=1;i<=3;i++)
  for(j=1;j<=9;j++){
fprintf(rightf2D,"%.5f    %.5f\n",MyCentres[i].RPoints[j].x,MyCentres[i].RPoints[j].y);
fprintf(leftf2D,"%.5f    %.5f\n",MyCentres[i].LPoints[j].x,MyCentres[i].LPoints[j].y);
```

```
fprintf(f3D,"%.5f    %.5f
%.5f\n",MyCentres[i].WPoints[j].x,MyCentres[i].WPoints[j].y,MyCentres[i].WPoints[j].z);
}
fclose(rightf2D);
fclose(f3D);
fclose(leftf2D);
}

void CcalibrbView::OnCalibrateStart()
{return;
  int i=1;
// if(flag3==0) {MessageBox("finish calibration first"); return;}
// if (flag3==2)
  {
S_IMAGE[i].RImage.DeleteObject( );
S_IMAGE[i].LImage.DeleteObject( );
S_IMAGE[i].TRImage.DeleteObject( );
S_IMAGE[i].TLImage.DeleteObject( );
S_IMAGE[i].CRImage.DeleteObject( );
S_IMAGE[i].CLImage.DeleteObject( );
  }//free buffers
  camera mycam;
  mycam.initialize();
  mycam.GRAB();mycam.GRAB();mycam.GRAB();
//reusing the buffers used for calibration, plane 1

S_IMAGE[i].width=mycam.S_X;
S_IMAGE[i].height=mycam.S_Y;
S_IMAGE[i].RImage.Attach(mycam.ArrayToBitmap(mycam.userImage1));
S_IMAGE[i].LImage.Attach(mycam.ArrayToBitmap(mycam.userImage2));
S_IMAGE[i].TRImage.Attach(mycam.ArrayToBitmap(mycam.userImage1));
S_IMAGE[i].TLImage.Attach(mycam.ArrayToBitmap(mycam.userImage2));
S_IMAGE[i].CRImage.Attach(mycam.ArrayToBitmap(mycam.userImage1));
S_IMAGE[i].CLImage.Attach(mycam.ArrayToBitmap(mycam.userImage2));
mycam.export2();


CORRESPOND dlg1;
dlg1.load(&S_IMAGE[i].LImage,&S_IMAGE[i].RImage,FL,FR,ML,MR,mycam.S_Y,mycam.S_X
);
dlg1.DoModal ();
mycam.release();
flag3=2; //indicate buffer is full now

}

void CcalibrbView::OnRobotReset()
{
  robot myrob;
myrob.initilize();
myrob.reset();
myrob.closecomm();
```

```
}

void CcalibrbView::OnRobotMoveaside()
{
  robot myrob;
myrob.initilize();
myrob.home2();
myrob.closecomm();

}
// calibrbView.h : interface of the CcalibrbView class
//
/////////////////////////////////////////////////////////////////////////////
//
#include "camera.h"
#include "aux2.h"
#include "Order.h"
#if
!defined(AFX_calibrbVIEW_H__7E3B7836_7668_4A7E_B51E_CB57750CBA8C__INCLUDE
D_)
#define
AFX_calibrbVIEW_H__7E3B7836_7668_4A7E_B51E_CB57750CBA8C__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
  typedef struct {
        CBitmap RImage;
        CBitmap LImage;
      CBitmap TRImage;
        CBitmap TLImage;
      CBitmap CRImage;
        CBitmap CLImage;
        int width;
        int height;
  } IMAGE;

class CcalibrbView : public CScrollView
{
protected: // create from serialization only
  CcalibrbView();
  DECLARE_DYNCREATE(CcalibrbView)

// Attributes
public:
  CcalibrbDoc* GetDocument();

// Operations
public:
```

```
CENTRES MyCentres[4];
CENTRES MyCentres_real[4];

IMAGE S_IMAGE[4];
int flag1;
//normalization parameters



double* cam_calibL;
double* cam_calibR;
double** FL;//fundamental matrix left
double** FR;//fundamental matrix right
double** ML;//fundamental matrix left
double** MR;//fundamental matrix right

int level;
double left_x_error;
double left_y_error;
double right_x_error;
double right_y_error;
int flag3;//this flag indicates when calibration is finished


int calibrate_flag;
// Overrides
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CcalibrbView)
  public:
  virtual void OnDraw(CDC* pDC);  // overridden to draw this view
  virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
  protected:
  virtual void OnInitialUpdate(); // called first time after construct
  virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
  virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
  virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
  //}}AFX_VIRTUAL

// Implementation
public:
  void make2D(double a[],double** b);
  void writecentres(CDC *pDC);
  int flag_centres;
  virtual ~CcalibrbView();
#ifdef _DEBUG
  virtual void AssertValid() const;
  virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
```

```
protected:
  //{{AFX_MSG(CcalibrbView)
  afx_msg void OnCameraGrab();
  afx_msg void OnCameraCalibrate();
  afx_msg void OnShowLeftimage();
  afx_msg void OnShowRightimage();
  afx_msg void OnShowTleftimage();
  afx_msg void OnShowTrightimage();
  afx_msg void OnShowLeftborder();
  afx_msg void OnShowRightborder();
  afx_msg void OnCameraReleaseandclose();
  afx_msg void OnShowShowcentres();
  afx_msg void OnCameraShowcalibsteps();
  afx_msg void OnLevel1();
  afx_msg void OnLevel2();
  afx_msg void OnLevel3();
  afx_msg void OnCalibrateNest();
  afx_msg void OnCalibrateCalibrate();
  afx_msg void OnRobotHome();
  afx_msg void OnRobotOpen();
  afx_msg void OnRobotPutback();
  afx_msg void OnDebugWritepoints();
  afx_msg void OnCalibrateStart();
  afx_msg void OnRobotReset();
  afx_msg void OnRobotMoveaside();
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
private:
  camera MyCam;
  aux2 calibrate;
  CBitmap* curb;
};

#ifndef _DEBUG  // debug version in calibrbView.cpp
inline CcalibrbDoc* CcalibrbView::GetDocument()
   { return (CcalibrbDoc*)m_pDocument; }
#endif

/////////////////////////////////////////////////////////////////////////////
//

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the previous line.

#endif //
!defined(AFX_calibrbVIEW_H__7E3B7836_7668_4A7E_B51E_CB57750CBA8C__INCLUDE
D_)
```

// robthirdView.cpp : implementation of the CrobthirdView class
//class for stereo matching using HNN

```
#include "stdafx.h"

#include "robthird.h"
#include "MATX.h"
#include <process.h>
#include "math.h"
#include "robthirdDoc.h"
#include "robot.h"
#include "robthirdView.h"
#include "HStereo.h"
#include "HFunctiongraph.h"
#include "MainFrm.h"

#include "Hopdense.h"

#define X_OFFSET 0
#define Y_OFFSET 30
#define X_GAP 30
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
#define THRESHOLD 1
#define SIZELIMIT 30
#define PI 3.14159265
/////////////////////////////////////////////////////////////////////////////
//
// CrobthirdView

IMPLEMENT_DYNCREATE(CrobthirdView, CFormView)

BEGIN_MESSAGE_MAP(CrobthirdView, CFormView)
  //{{AFX_MSG_MAP(CrobthirdView)
  ON_COMMAND(ID_MODE_FILE, OnModeFile)
  ON_COMMAND(ID_MODE_CAMERA, OnModeCamera)
  ON_COMMAND(ID_RECONSTRUCTION_LOAD, OnReconstructionLoad)
  ON_COMMAND(ID_ZOOM_ZOOMOUT, OnZoomZoomout)
  ON_COMMAND(ID_ZOOM_ZOMIN, OnZoomZomin)
  ON_COMMAND(ID_ZOOM_RESET, OnZoomReset)
  ON_COMMAND(ID_FEATURES_SOBEL, OnFeaturesSobel)
  ON_COMMAND(ID_FEATURES_SUSANCORNERS, OnFeaturesSusancorners)
  ON_COMMAND(ID_FEATURES_SUSANEDGES, OnFeaturesSusanedges)
  ON_COMMAND(ID_FEATURES_MORAVEC, OnFeaturesMoravec)
  ON_COMMAND(ID_FEATURES_CANNY, OnFeaturesCanny)
  ON_COMMAND(ID_FEATURES_RESET, OnFeaturesReset)
  ON_COMMAND(ID_FEATURES_REMOVEBACKGROUND,
OnFeaturesRemovebackground)
  ON_COMMAND(ID_FEATURES_CHANGEDISPLAYSTYLE,
OnFeaturesChangedisplaystyle)
  ON_COMMAND(ID_MATCHING_MATCHLINES, OnMatchingMatchlines)
```

```
  ON_WM_LBUTTONDOWN()
  ON_COMMAND(ID_MATCHING_MATCHALL, OnMatchingMatchall)
  ON_COMMAND(ID_FEATURES_REDUCEFEATURES, OnFeaturesReducefeatures)
  ON_WM_VSCROLL()
  ON_WM_HSCROLL()
  ON_COMMAND(ID_MATCHING_MATCHALL2, OnMatchingMatchall2)
  ON_COMMAND(ID_CHECK_CHECK, OnCheckCheck)
  ON_COMMAND(ID_SHOWPOINTS_SHOWNEXTPOINT, show_differentvalue)
  ON_COMMAND(ID_SHOWPOINTS_RESET, unshow_value)
  ON_COMMAND(ID_PRINTDATA_PRINTFEATURES, OnPrintdataPrintfeatures)
  ON_COMMAND(ID_SHOWPOINTS_SHOWDISPAND3D,
OnShowpointsShowdispand3d)
  ON_COMMAND(ID_SHOWPOINTS_NEWWINDOWFORSTATS,
OnShowpointsNewwindowforstats)
  ON_COMMAND(ID_SHOWPOINTS_SHOWPROJECTION,
OnShowpointsShowprojection)
  ON_COMMAND(ID_ROBOT_MOVEASIDE, OnRobotMoveaside)
  ON_COMMAND(ID_ROBOT_GRAB1, OnRobotGrab1)
  ON_COMMAND(ID_ROBOT_OPEN, OnRobotOpen)
  ON_COMMAND(ID_ROBOT_NEST, OnRobotNest)
  ON_COMMAND(ID_ROBOT_HOME, OnRobotHome)
  ON_COMMAND(ID_ROBOT_RESET, OnRobotReset)
  ON_COMMAND(ID_ROBOT_CALIBRATIONPOSITION1,
OnRobotCalibrationposition1)
  ON_COMMAND(ID_ROBOT_INTELLICAM, OnRobotIntellicam)
  ON_COMMAND(ID_ROBOT_PUTBACK, OnRobotPutback)
  ON_COMMAND(ID_ROBOT_GRABTARGETS, grab1)
  ON_COMMAND(ID_MATCHING_MATCHHOPFIELD, OnMatchingMatchhopfield)
  ON_COMMAND(ID_MATCHING_HOPFIELDTEST1, OnMatchingHopfieldtest1)
  ON_COMMAND(ID_CHECK_SHOWDISPARITYGRADIENT,
OnCheckShowdisparitygradient)
  ON_COMMAND(ID_MATCHING_HOPFIELDMATCH, OnMatchingHopfieldmatch)
  ON_WM_MOUSEMOVE()
  ON_COMMAND(ID_GRAPH_GRAPH1, OnGraphGraph1)
  ON_COMMAND(ID_CHECK_CHECKHRBITMAP, OnCheckCheckhrbitmap)
  ON_COMMAND(ID_MATCHING_DENSEMATCH1, OnMatchingDensematch1)
  ON_COMMAND(ID_CHECK_INCREASECONTRASTOFIMAGE1,
OnCheckIncreasecontrastofimage1)
  ON_COMMAND(ID_MATCHING_DENSEMATCHCORRELATION,
OnMatchingDensematchcorrelation)
  ON_COMMAND(ID_CHECK_ENABLELINEMATCHING, OnCheckEnablelinematching)
  ON_COMMAND(ID_CHECK_FORCERELOADIMAGE1, OnCheckForcereloadimage1)
  ON_COMMAND(ID_CHECK_FORCERELOADIMAGE2, OnCheckForcereloadimage2)
  ON_COMMAND(ID_FEATURES_INCREASESENSITIVITY,
OnFeaturesIncreasesensitivity)
  ON_COMMAND(ID_FEATURES_DECREASESENSITIVITY,
OnFeaturesDecreasesensitivity)
  ON_COMMAND(ID_FEATURES_RESETSENSITIVITY, OnFeaturesResetsensitivity)
  ON_COMMAND(ID_MATCHING_FULLDENSEMATCHINGCORRELATION,
OnMatchingFulldensematchingcorrelation)
  ON_COMMAND(ID_CHECK_SHOWGROUNDTRUTH, OnCheckShowgroundtruth)
  //}}AFX_MSG_MAP
```

```
  // Standard printing commands
  ON_COMMAND(ID_FILE_PRINT, CFormView::OnFilePrint)
  ON_COMMAND(ID_FILE_PRINT_DIRECT, CFormView::OnFilePrint)
  ON_COMMAND(ID_FILE_PRINT_PREVIEW, CFormView::OnFilePrintPreview)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
//
// CrobthirdView construction/destruction

CrobthirdView::CrobthirdView()
  : CFormView(CrobthirdView::IDD)
{
  //{{AFX_DATA_INIT(CrobthirdView)
    // NOTE: the ClassWizard will add member initialization here
  //}}AFX_DATA_INIT
  // TODO: add construction code here

}

CrobthirdView::~CrobthirdView()
{
}

void CrobthirdView::DoDataExchange(CDataExchange* pDX)
{
  CFormView::DoDataExchange(pDX);
  //{{AFX_DATA_MAP(CrobthirdView)
    // NOTE: the ClassWizard will add DDX and DDV calls here
  //}}AFX_DATA_MAP
}

BOOL CrobthirdView::PreCreateWindow(CREATESTRUCT& cs)
{
  // TODO: Modify the Window class or styles here by modifying
  //  the CREATESTRUCT cs

  return CFormView::PreCreateWindow(cs);
}

void CrobthirdView::OnInitialUpdate()
{


//phi[1]=0;
  CFormView::OnInitialUpdate();
  GetParentFrame()->RecalcLayout();
  ResizeParentToFit();
////////////////////////////////
  modeflag=0;
  zoomflag=1;
  imagesloaded=0;
```

```
 background=1;
 match_lines=0;
 F_matrix_loaded=0;
 match_array_loaded=0;
 sensitivity=50;
 disparity_gradient_demo=0;
 ipshow_value=0;
 showprojection=0;
 enable_line_match=0;
 for(int i=0;i<20;i++) {surfaces[i].size=0; surfaces[i].value=0;}
OnReconstructionLoad() ;
}

/////////////////////////////////////////////////////////////////////////////
//
// CrobthirdView printing

BOOL CrobthirdView::OnPreparePrinting(CPrintInfo* pInfo)
{
 // default preparation
 return DoPreparePrinting(pInfo);
}

void CrobthirdView::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
 // TODO: add extra initialization before printing
}

void CrobthirdView::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
 // TODO: add cleanup after printing
}

void CrobthirdView::OnPrint(CDC* pDC, CPrintInfo* /*pInfo*/)
{
 // TODO: add customized printing code here
}

/////////////////////////////////////////////////////////////////////////////
//
// CrobthirdView diagnostics

#ifdef _DEBUG
void CrobthirdView::AssertValid() const
{
 CFormView::AssertValid();
}

void CrobthirdView::Dump(CDumpContext& dc) const
{
 CFormView::Dump(dc);
}
```

```
CrobthirdDoc* CrobthirdView::GetDocument() // non-debug version is inline
{
 ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CrobthirdDoc)));
 return (CrobthirdDoc*)m_pDocument;
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
//
// CrobthirdView message handlers

void CrobthirdView::OnModeFile()
{
modeflag=0;
}

void CrobthirdView::OnModeCamera()
{
modeflag=1;
 }

void CrobthirdView::OnReconstructionLoad()
{
 if(modeflag==1)
 {
 cleandirectory();
 GRAB();Sleep(8000);
 }


if(imagesloaded==1) return;

//image[0].open("images\\truedisp.bmp");
image[1].open("..\\images\\right.bmp");
image[2].open("..\\images\\left.bmp");


fixarea();
imagesloaded=1;


Invalidate();
}

void CrobthirdView::cleandirectory()
{
DeleteFile("..\\images\\left.bmp");
DeleteFile("..\\images\\right.bmp");

}
```

```
void CrobthirdView::GRAB()
{

 HINSTANCE hInst = ShellExecute(0,
                        "open",                 // Operation to perform
                        "camera.exe",  // Application name
                        "images",          // Additional parameters
                        0,                        // Default directory
                        SW_HIDE );
 }
void CrobthirdView::changebackdir()
{
char   szAppPath[MAX_PATH] = "";
CString strAppDirectory;

::GetModuleFileName(0, szAppPath, sizeof(szAppPath) - 1);

// Extract directory
strAppDirectory = szAppPath;
strAppDirectory = strAppDirectory.Left(strAppDirectory.ReverseFind('\\'));
SetCurrentDirectory(strAppDirectory.GetBuffer(strAppDirectory.GetLength()));

}

void CrobthirdView::fixarea()
{


rc1.top=Y_OFFSET;
rc1.bottom=rc1.top+(int)((float)image[1].height/zoomflag);
rc1.left=X_OFFSET;
rc1.right=rc1.left+(int)((float)image[1].width/zoomflag);
//first rectangle

rc2.top=Y_OFFSET;
rc2.bottom=rc2.top+(int)((float)image[2].height/zoomflag);
rc2.left=rc1.right+X_GAP;
rc2.right=rc2.left+(int)((float)image[2].width/zoomflag);

//rc3 holds most of the screen, for the purpose of clearing leaked crosses
rc3.top=rc1.top;
rc3.left=rc1.left;
rc3.right=rc2.right;
rc3.bottom=rc2.bottom;
////////////////////////////////////////////////////////////////////////////

CMainFrame* pMainFrame = (CMainFrame*) ::AfxGetMainWnd();
pMainFrame->resize(&rc3);

}

void CrobthirdView::OnZoomZoomout()
```

```
{
if(imagesloaded==0) return ;
  zoomflag*=1.2;

  fixarea();
Invalidate();

}

void CrobthirdView::OnZoomZomin()
{
if(imagesloaded==0) return ;
  zoomflag/=1.2;

  fixarea();
Invalidate();

}

void CrobthirdView::OnZoomReset()
{
if(imagesloaded==0) return ;
  zoomflag=1.000;

  fixarea();
Invalidate();

}

void CrobthirdView::writetitle(CDC* pDC)
{
CFont newFont;
newFont.CreateFont(18,9,0,0,0,0,0,0,0,0,OUT_DEFAULT_PRECIS,CLIP_DEFAULT_PRECIS,P
ROOF_QUALITY,FF_DONTCARE,"Arial");
pDC->SelectObject(&newFont);


int x1,y1,x2,y2;

x1=(int)(180/zoomflag);
y1=5;

x2=(int)(900/zoomflag);
y2=5;

pDC->TextOut(x1,y1,"Left Camera");
pDC->TextOut(x2,y2,"Right Camera");

}

void CrobthirdView::OnFeaturesSobel()
{
```

```
    CF[1].edge_display_flag=2;
    CF[2].edge_display_flag=2;
    reset_display();
CF[1].sobeledge(&image[1]);
CF[2].sobeledge(&image[2]);

  Invalidate();
}

void CrobthirdView::OnFeaturesSusancorners()
{
    CF[1].edge_display_flag=3;
    CF[2].edge_display_flag=3;
    reset_display();
CF[1].susan_corner(&image[1]);
CF[2].susan_corner(&image[2]);

  Invalidate();

}

void CrobthirdView::OnFeaturesSusanedges()
{
    CF[1].edge_display_flag=2;
    CF[2].edge_display_flag=2;
    reset_display();
CF[1].susan_edge(&image[1]);
CF[2].susan_edge(&image[2]);
  Invalidate();

  }

void CrobthirdView::OnFeaturesMoravec()
{

    CF[1].edge_display_flag=3;
    CF[2].edge_display_flag=3;

    reset_display();
CF[1].moravec(&image[1]);
CF[2].moravec(&image[2]);

  Invalidate();
  }

void CrobthirdView::OnFeaturesCanny()
{
    CF[1].edge_display_flag=2;
    CF[2].edge_display_flag=2;

    reset_display();
 CF[1].canny(&image[1]);
```

```
CF[2].canny(&image[2]);

  Invalidate();
}

void CrobthirdView::OnFeaturesReset()
{
reset_display();
CF[1].features.num=0;
CF[2].features.num=0;
      CF[1].edge_display_flag=2;
    CF[2].edge_display_flag=2;

  Invalidate();

}

void CrobthirdView::OnFeaturesRemovebackground()
{
  if(background==1) background=0;
  else background=1;
  Invalidate();

}

void CrobthirdView::OnFeaturesChangedisplaystyle()
{
  if(CF[1].edge_display_flag==0) {
    CF[1].edge_display_flag=1;
    CF[2].edge_display_flag=1;
  } else if(CF[1].edge_display_flag==1) {
    CF[1].edge_display_flag=2;
    CF[2].edge_display_flag=2;
  }
  else if(CF[1].edge_display_flag==2) {
    CF[1].edge_display_flag=3;
    CF[2].edge_display_flag=3;
  }
  else{
    CF[1].edge_display_flag=0;
    CF[2].edge_display_flag=0;

  }
  Invalidate();
}

void CrobthirdView::OnMatchingMatchlines()
{
match_lines=1;
}

void CrobthirdView::OnLButtonDown(UINT nFlags, CPoint point)
```

```
{
 // TODO: Add your message handler code here and/or call default
 //createDGPoints(point);
 //if(match_lines==1) matchlines(point);


 if(ipshow_value!=0 && rc1.PtInRect(point)){
   {

   CPoint zpoint;

     zpoint.y=point.y-rc1.top;
     zpoint.y*=zoomflag;


   if(zpoint.y>CF[1].features.y[ipshow_value])
     {
     while(CF[1].features.y[ipshow_value]<zpoint.y)
     ipshow_value++;
     }
   else
     {
     while(CF[1].features.y[ipshow_value]>zpoint.y)
     ipshow_value--;

     }

   show_differentvalue();

   }
   Invalidate();
   }


 if(rc1.PtInRect(point) || rc2.PtInRect(point))
 {
 if(enable_line_match==1)
   {

OnReconstructionLoad() ;
reset_display();


CPoint zpoint;
zpoint.y=point.y-rc1.top;
zpoint.y*=zoomflag;



  // TODO: Add your command handler code here
```

```
  CHopdense dense(&image[1],&image[2],&CF[1],&CF[2]);
  dense.dense_match_1(2,zpoint.y);
    CF[1].edge_display_flag=1;
    CF[2].edge_display_flag=1;
    enable_line_match=0;
    OnCheckForcereloadimage1();
Invalidate();
    }
  }

  CFormView::OnLButtonDown(nFlags, point);
}


void CrobthirdView::load_F_matrix(void)
{
CFun.loadmatrix();
F_matrix_loaded=1;
}

CPoint CrobthirdView::fixpoint(CPoint p, CRect rc, float zoom)
{
return 0;
}

void CrobthirdView::OnMatchingMatchall()
{
matchall();
}

int CrobthirdView::matchall()
{

  int windowsize=9;
//free if allocated
  int num_points;
  int order,other;
  if(match_array_loaded==1)
  delete [] matches;

  if(CF[1].features.num>CF[2].features.num)
    order=2;
  else
    order=1;

  //choose the image with the smaller number of feature points
  if(order==1) {num_points=CF[1].features.num; other=2;}
  else {num_points=CF[2].features.num; other=1;}
  matches=new CPoint[num_points];//allocate memory

  int i;
CPoint pp;
```

```
CFun.load_array_tracker(num_points);
  for(i=1;i<=num_points;i++){
    pp.x=CF[order].features.x[i];
    pp.y=CF[order].features.y[i];

    matches[i].x=i;
    matches[i].y=CFun.ZNCC(&image[order],&image[other],windowsize,pp,&CF[other],order);
  }

///write matches to text file
FILE* tfile;
tfile=fopen("matches.txt","wt");
fprintf(tfile,"order number =%d  and other number =%d\n",num_points,
CF[other].features.num);
  for(i=1;i<=num_points;i++){
    fprintf(tfile,"%d   ----------   %d\n",matches[i].x,matches[i].y);
    }
fclose(tfile);


int* xq=new int[num_points+2];
int* yq=new int[num_points+2];
int cc1=0;//counter for the following loop
//remove any points in the original if no correspondence was found

  for(i=1;i<=num_points;i++){
    if(matches[i].y!=-1){
    xq[i]=CF[order].features.x[i];
    yq[i]=CF[order].features.y[i];
      cc1++;
      }
    }



  for(i=1;i<=cc1;i++){
    CF[order].features.x[i]=xq[i];
    CF[order].features.y[i]=yq[i];
  }

CF[order].features.num=cc1;//change size of the structure

int cc2=0;

  for(i=1;i<=num_points;i++){
    if(matches[i].y!=-1){
    xq[i]=CF[other].features.x[matches[i].y];
    yq[i]=CF[other].features.y[matches[i].y];
      cc2++;
      }
  }
```

```
  for(i=1;i<=cc2;i++){
    CF[other].features.x[i]=xq[i];
    CF[other].features.y[i]=yq[i];
  }
CF[other].features.num=cc2;//change size of the structure
delete [] xq;
delete [] yq;


Invalidate();
return 0;
}
int CrobthirdView::matchall_1()
{int j;
  int windowsize=9;

//free if allocated
  int num_points;
  int order,other;
  if(match_array_loaded==1)
  delete [] matches;

  if(CF[1].features.num>CF[2].features.num)
    order=2;
  else
    order=1;

  //chpoose the image with the smaller number of feature points
  if(order==1) {num_points=CF[1].features.num; other=2;}
  else {num_points=CF[2].features.num; other=1;}
  matches=new CPoint[num_points];//allocate memory
//form correlation value grid
float** match_grid=matrix(CF[order].features.num,CF[other].features.num);


  int i;
CPoint pp;
CFun.load_array_tracker(num_points);
  for(i=1;i<=num_points;i++){
    pp.x=CF[order].features.x[i];
    pp.y=CF[order].features.y[i];
    CFun.ZNCC_1(&image[order],&image[other],windowsize,pp,&CF[other],order,match_grid[i]);
  }
//writematrix(match_grid,CF[order].features.num,CF[other].features.num,"gridbefore.txt");
clean_grid(match_grid,CF[order].features.num,CF[other].features.num);//remove ambiguity
//writematrix(match_grid,CF[order].features.num,CF[other].features.num,"gridafter.txt");

//from the grid, for matches
for(i=1;i<=CF[order].features.num;i++){
  matches[i].x=i;matches[i].y=-1;
  for(j=1;j<=CF[other].features.num;j++){
if(match_grid[i][j]!=0) matches[i].y=j;
```

```
}}


///write matches to text file
FILE* tfile;
tfile=fopen("matches.txt","wt");
fprintf(tfile,"order number =%d  and other number =%d\n",num_points,
CF[other].features.num);
  for(i=1;i<=num_points;i++){
    fprintf(tfile,"%d   ----------   %d\n",matches[i].x,matches[i].y);
    }
fclose(tfile);


int* xq=new int[num_points+2];
int* yq=new int[num_points+2];
int cc1=1;//counter for the following loop
//remove any points in the original if no correspondence was found

  for(i=1;i<=num_points;i++){
    if(matches[i].y!=-1){//check for invalid Zs
  /* CPoint p1,p2;
p1.x= cf1->features.x[i]  ;
p1.y= cf1->features.y[i] ;
p2.x=cf2->features.x[i]   ;
p2.y=cf2->features.y[i]   ;
this->reconstruct(p1,p2,1,&cf1->worldpts.x[i],&cf1->worldpts.y[i],&cf1->worldpts.z[i]);*/
    xq[cc1]=CF[order].features.x[i];
    yq[cc1]=CF[order].features.y[i];
      cc1++;
      }
    }



  for(i=1;i<cc1;i++){
    CF[order].features.x[i]=xq[i];
    CF[order].features.y[i]=yq[i];
  }

CF[order].features.num=cc1-1;//change size of the structure

int cc2=1;

  for(i=1;i<=num_points;i++){
    if(matches[i].y!=-1){
    xq[cc2]=CF[other].features.x[matches[i].y];
    yq[cc2]=CF[other].features.y[matches[i].y];
      cc2++;
      }
  }
```

```
 for(i=1;i<cc2;i++){
   CF[other].features.x[i]=xq[i];
   CF[other].features.y[i]=yq[i];
 }
CF[other].features.num=cc2-1;//change size of the structure
delete [] xq;
delete [] yq;
///find 3D points and disparitiers, this line is important

CFun.find3D_and_disp(&CF[1],&CF[2]);
CF[1].calc_statistics();
findorientation();//find centroid
Invalidate();
return 0;
}
void CrobthirdView::OnFeaturesReducefeatures()
{
CF[1].reduce();
CF[2].reduce();
Invalidate();
}

void CrobthirdView::OnVScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{
 // TODO: Add your message handler code here and/or call default

 CFormView::OnVScroll(nSBCode, nPos, pScrollBar);
}

void CrobthirdView::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{
 // TODO: Add your message handler code here and/or call default

 CFormView::OnHScroll(nSBCode, nPos, pScrollBar);
}

void CrobthirdView::clean_grid(float **pgrid,int m,int n)
{
int i,j;
float max;
int index;



 //find max along the row

for(i=1;i<=m;i++){
 max=-10000;
 //find biggest value
for(j=1;j<=n;j++){
 if(pgrid[i][j]>max && pgrid[i][j]!=0){
```

```
 max=pgrid[i][j];

 index=j;
 }}
 //set everything else to zero
for(j=1;j<=n;j++){
 if(j!=index){
 pgrid[i][j]=0;
   }}


 }

for(j=1;j<=n;j++){
 max=-10000;
 //find biggest value
for(i=1;i<=m;i++){
 if(pgrid[i][j]>max && pgrid[i][j]!=0){
 max=pgrid[i][j];
 pgrid[i][j]=1;
 index=i;
 }}
 //set everything else to zero
for(i=1;i<=m;i++){
 if(i!=index){
 pgrid[i][j]=0;
   }}


 }


}

void CrobthirdView::OnMatchingMatchall2()
{
matchall_1();
}

void CrobthirdView::OnDraw(CDC* pDC)
{
 if(F_matrix_loaded==0) load_F_matrix();


 if(imagesloaded==1 && background==1)
 {
 pDC->SetStretchBltMode( HALFTONE);
 CDC memdc1;
 memdc1.CreateCompatibleDC(pDC);
 image[1].draw(&memdc1);
 pDC-
>StretchBlt(rc1.left,rc1.top,rc1.Width(),rc1.Height(),&memdc1,0,0,image[1].width,image[1].height,
```

```
SRCCOPY );
  CDC memdc2;
  memdc2.CreateCompatibleDC(pDC);
  image[2].draw(&memdc2);


  pDC-
>StretchBlt(rc2.left,rc2.top,rc2.Width(),rc2.Height(),&memdc2,0,0,image[2].width,image[2].height,
SRCCOPY );
  drawDGdemopts(pDC);
// this->writetitle(pDC);
  }
  //epipolar thing,
  if(ipshow_value>0){//draw epipolar lines
    if(ipshow_value>CF[1].features.num) ipshow_value=1;
  CPoint tpoint;tpoint.x=CF[1].features.x[ipshow_value];tpoint.y=CF[1].features.y[ipshow_value];
  //CFun.epipolar_line(tpoint,&image[1],pDC,zoomflag,rc2,&CF[2]);
  }

if(showprojection==0)
  {
  CF[1].draw(pDC,&rc1,zoomflag);
  CF[2].draw(pDC,&rc2,zoomflag);
  }

if(showprojection==1) drawprojection(pDC);

}

void CrobthirdView::OnCheckCheck()
{
  int i,j;
long float a1,a2;
a1=a2=0;
OnReconstructionLoad() ;//get images

for (i = 0; i < image[1].height ; i++) {
    for (j = 0; j < image[1].width; j++) {

    a1+=image[1].getpixel(j,i);
     a2+=image[2].getpixel(j,i);

   }
  }
long float a3,a4;
a3=a1-a2;
a4=image[1].height*image[1].width;
a3=a3/a4;


  CString b;
  b.Format("A= %f B=%f and d=%f",a1,a2,a3);
```

```
 MessageBox(b);

 }

BOOL CrobthirdView::PreTranslateMessage(MSG* pMsg)
{
   if (pMsg->message ==WM_KEYUP){
if(pMsg->wParam==189) OnZoomZoomout() ;
if(pMsg->wParam==187) OnZoomZomin() ;
if(pMsg->wParam==48) OnZoomReset() ;
if(pMsg->wParam==87) OnZoomReset() ;
if(pMsg->wParam==76) OnReconstructionLoad() ;
if(pMsg->wParam==67) OnFeaturesChangedisplaystyle() ;
if(pMsg->wParam==86) show_differentvalue() ;
if(pMsg->wParam==66) unshow_value() ;
if(pMsg->wParam==88) performfull1();


 }

 return CFormView::PreTranslateMessage(pMsg);
}

void CrobthirdView::show_differentvalue()
{

ipshow_value++;
CF[1].edge_display_flag=1;
CF[2].edge_display_flag=1;

CF[1].showval=ipshow_value;
CF[2].showval=ipshow_value;
CF[1].show3d=1;
CF[2].show3d=1;
showprojection=0;

CPoint zpoint;
zpoint.x=CF[1].features.x[ipshow_value];
zpoint.y=CF[1].features.y[ipshow_value];
int disp=CF[2].features.x[ipshow_value]-CF[1].features.x[ipshow_value];

 CMainFrame *pMainWnd = (CMainFrame *)AfxGetMainWnd();
  CString s;
  s.Format("X=%d Y=%d  Disparity=%d",zpoint.x,zpoint.y,disp);
   pMainWnd->m_wndStatusBar.SetPaneText(0,s);



Invalidate();
}
```

```
void CrobthirdView::unshow_value()
{
ipshow_value=0;
CF[1].showval=ipshow_value;
CF[2].showval=ipshow_value;
CF[1].show3d=0;
CF[2].show3d=0;
Invalidate();
}
void CrobthirdView::OnPrintdataPrintfeatures()
{
CF[1].printfeatures("features1.txt");
CF[2].printfeatures("features2.txt");

}

void CrobthirdView::OnShowpointsShowdispand3d()
{

if(CF[1].show3d==0){
CF[1].show3d=1;
CF[2].show3d=1;

  } else {

CF[1].show3d=0;
CF[2].show3d=0;

  }



Invalidate();
}

void CrobthirdView::reset_display()
{
CF[1].show3d=0;
CF[2].show3d=0;
ipshow_value=0;
CF[1].showval=ipshow_value;
CF[2].showval=ipshow_value;



}


void CrobthirdView::OnShowpointsNewwindowforstats()
{
if(CF[1].window_or_this==0){
```

```
CF[1].window_or_this=1;
  CF[1].dispinfo.Create(IDD_DIALOG1, AfxGetMainWnd());
CF[1].dispinfo.ShowWindow(SW_SHOW);
AfxGetMainWnd()->SetFocus();}
else {
CF[1].window_or_this=0;
CF[1].dispinfo.DestroyWindow( );}

Invalidate();
}

void CrobthirdView::findorientation()
{
int i,j;


int flag=0;
int usedclusters=1;

for(i=0;i<40;i++){
  surfaces[i].size=0;
  surfaces[i].value=0;
  }

//find how many objects with rhe same height
//also find their mean centroi on the surface
for(i=1;i<=CF[1].features.num;i++)
  {

  if(CF[1].worldpts.z[i]>1 && CF[1].worldpts.z[i]<15)
    {
    flag=0;
    for(j=1;j<usedclusters;j++)
      {
      if(fabs(CF[1].worldpts.z[i]-surfaces[j].value)<THRESHOLD)
        {
        addCluster(CF[1].worldpts.z[i],j);
        flag=1; break;
        }
      }
      if(flag==0)
        {
        addCluster(CF[1].worldpts.z[i],j);
        usedclusters++;
        if(usedclusters>38) {MessageBox("Levels exceeded"); return;}
        }

    }

  }
```

```
//merge groups that are alike
for(j=1;j<usedclusters;j++){
  for(i=1;i<usedclusters;i++){
    if(i!=j){
      if(fabs(surfaces[j].value-surfaces[i].value)<THRESHOLD)
        {

  surfaces[j].value=(surfaces[j].value*surfaces[j].size+surfaces[i].value*surfaces[i].size)/(surfaces[j].s
ize+surfaces[i].size);
        surfaces[j].size+=surfaces[i].size;
        surfaces[i].size=0;
  }}}}

//find centroid

      for(i=0;i<20;i++) {centres[i].x=centres[i].y=centres[i].z=0;}

//find Z coordinates of the centroids
int counter=1;
for(i=1;i<usedclusters;i++){
  if(surfaces[i].size>SIZELIMIT)
    {
    centres[counter].z=surfaces[i].value+0.2;//all depths seem to be 0.2 short, so here im adding
0.2
    counter++;
  }}

numObjects=counter-1;//we found number of objects
//find X and Y coordinates of the centroids
  float temp_counter[20];
  for(i=0;i<20;i++) temp_counter[i]=0;


  for(i=1;i<=CF[1].features.num;i++){
    for(j=1;j<=numObjects;j++){
    if(fabs(CF[1].worldpts.z[i]-centres[j].z)<THRESHOLD){
    //this keeps the number of points already added, used to find the average
    temp_counter[j]++;

    if(centres[j].x==0)
      centres[j].x=CF[1].worldpts.x[i];
    else
      centres[j].x=(centres[j].x*temp_counter[j]+CF[1].worldpts.x[i])/(temp_counter[j]+1);

    if(centres[j].y==0)
      centres[j].y=CF[1].worldpts.y[i];
    else
      centres[j].y=(centres[j].y*temp_counter[j]+CF[1].worldpts.y[i])/(temp_counter[j]+1);
```

```
    }}}
//now find the reprojection of the centroids on the first camera
    for(i=0;i<20;i++) {proj_centres[i].X=proj_centres[i].Y=0;}

    //multiuply centroid with projection matrix, reprojecting on image
    for(i=1;i<=numObjects;i++)
      {

  proj_centres[i].X=CFun.M[1][1][1]*centres[i].x+CFun.M[1][1][2]*centres[i].y+CFun.M[1][1][3]*c
entres[i].z+CFun.M[1][1][4];

  proj_centres[i].Y=CFun.M[1][2][1]*centres[i].x+CFun.M[1][2][2]*centres[i].y+CFun.M[1][2][3]*c
entres[i].z+CFun.M[1][2][4];
      float
scale=CFun.M[1][3][1]*centres[i].x+CFun.M[1][3][2]*centres[i].y+CFun.M[1][3][3]*centres[i].z+C
Fun.M[1][3][4];
      //divide by scale
      proj_centres[i].X/=scale;
      proj_centres[i].Y/=scale;
      }
//find phi or axis of elongation
  findPhi2();
//now find the line on the image by choosing two points around the centroid and
//projecting them on the axis of elongation and then project on the image

    //initlize list of points
    for(i=0;i<20;i++){
      for(j=0;j<3;j++){
        phi_points_dc[i][j].X=phi_points_dc[i][j].Y=phi_points[i][j].X=phi_points[i][j].Y=0;}}

//finding where this line hits a circle to draw ther orientation line
float radius=3;
float m[20];
for(i=0;i<20;i++) m[i]=0;

for(j=1;j<=numObjects;j++)
m[j]=tan(phi[j]);

    for(j=1;j<=numObjects;j++)
      {

      phi_points[j][1].X=sqrt((radius*radius)/((m[j]*m[j])+1));

      if(phi[j]>(0.5*PI)) phi_points[j][1].X*=-1;

      phi_points[j][2].X=-phi_points[j][1].X;

      phi_points[j][1].Y=sqrt((radius*radius)-(phi_points[j][1].X*phi_points[j][1].X));
      phi_points[j][2].Y=-phi_points[j][1].Y;
```

```
    }

    float y1=(phi_points[1][1].X*phi_points[1][1].X)+(phi_points[1][1].Y*phi_points[1][1].Y);
    float y2=(phi_points[1][2].X*phi_points[1][2].X)+(phi_points[1][2].Y*phi_points[1][2].Y);

//now find their projection in the first image


    float tempx2,tempy2;

    for(i=1;i<=numObjects;i++){
      for(j=1;j<=2;j++)
      {
      phi_points[i][j].X+=centres[i].x;
      phi_points[i][j].Y+=centres[i].y;

      tempx2=phi_points[i][j].X;
      tempy2=phi_points[i][j].Y;


  phi_points[i][j].X=CFun.M[1][1][1]*tempx2+CFun.M[1][1][2]*tempy2+CFun.M[1][1][3]*centres
[i].z+CFun.M[1][1][4];

  phi_points[i][j].Y=CFun.M[1][2][1]*tempx2+CFun.M[1][2][2]*tempy2+CFun.M[1][2][3]*centres
[i].z+CFun.M[1][2][4];
      float
scale=CFun.M[1][3][1]*tempx2+CFun.M[1][3][2]*tempy2+CFun.M[1][3][3]*centres[i].z+CFun.M
[1][3][4];
      //divide by scale

      phi_points[i][j].X/=scale;
      phi_points[i][j].Y/=scale;
      //now create phi_points 2 for the device context
      phi_points_dc[i][j].X=phi_points[i][j].X;
      phi_points_dc[i][j].Y=phi_points[i][j].Y;

      phi_points_dc[i][j].X/=zoomflag;
      phi_points_dc[i][j].Y/=zoomflag;

      phi_points_dc[i][j].X+=rc1.left;
      phi_points_dc[i][j].Y+=rc1.top;

      }}

    wpoints2 centres_temp[20];
    for(i=1;i<=numObjects;i++)
    centres_temp[i]=centres[i];

    //have to sort the centres now
    for(i=1;i<=numObjects;i++)
      {
      float maxz=-1000;
```

```
    int index_temp=0;
    for(j=1;j<=numObjects;j++)
      {
      if(centres_temp[j].z>maxz)
        {
        maxz=centres_temp[j].z;
        index_temp=j;
        }
      }
      centres_sorted[i]=centres_temp[index_temp];
      centres_temp[index_temp].z=-100000;

    }

     }

void CrobthirdView::addCluster(float value, int K)
{
if(surfaces[K].size!=0){
float temp=surfaces[K].value*surfaces[K].size;
float temp2=(temp+value)/(surfaces[K].size+1);
surfaces[K].value=temp2;
  } else{
surfaces[K].value=value;

   }

  surfaces[K].size++;
}

void CrobthirdView::OnShowpointsShowprojection()
{
if(showprojection==0) showprojection=1;
else showprojection=0;

Invalidate();
}

void CrobthirdView::drawprojection(CDC *pDC)
{
int i=1;
CFont newFont;
CPoint cdots;
pDC->SetTextColor(RGB(200,0,0));
pDC->SetBkMode(TRANSPARENT);
newFont.CreateFont(12/zoomflag,8/zoomflag,0,0,0,0,0,0,OUT_DEFAULT_PRECIS,CLIP_D
EFAULT_PRECIS,PROOF_QUALITY,FF_DONTCARE,"Arial");
pDC->SelectObject(&newFont);

while(centres[i].z!=0)
  {
  cdots.x=proj_centres[i].X;
```

```
cdots.y=proj_centres[i].Y;
cdots.x/=zoomflag;
cdots.y/=zoomflag;
cdots.x+=rc1.left;
cdots.y+=rc1.top;

pDC->SetPixel(cdots,RGB(0,255,0));
pDC->SetPixel(cdots.x,cdots.y+1,RGB(0,255,0));
pDC->SetPixel(cdots.x,cdots.y-1,RGB(0,255,0));
pDC->SetPixel(cdots.x-1,cdots.y-1,RGB(0,255,0));
pDC->SetPixel(cdots.x+1,cdots.y+1,RGB(0,255,0));
pDC->SetPixel(cdots.x+1,cdots.y-1,RGB(0,255,0));
pDC->SetPixel(cdots.x-1,cdots.y+1,RGB(0,255,0));
pDC->SetPixel(cdots.x-1,cdots.y,RGB(0,255,0));
pDC->SetPixel(cdots.x+1,cdots.y,RGB(0,255,0));

//write text now
  CPen pen1;
pen1.CreatePen(PS_SOLID, 1, RGB(0,0,255));
CPen* pOldPen = pDC->SelectObject(&pen1);
CString b;
b.Format("X=%0.2f   Y=%0.2f   Z=%0.2f
phi=%0.2f",centres[i].x,centres[i].y,centres[i].z,phi[i]*180/PI);
pDC->TextOut(cdots.x-140,cdots.y-20,b);
//now draw the line
pDC->MoveTo(phi_points_dc[i][1].X,phi_points_dc[i][1].Y);
pDC->LineTo(phi_points_dc[i][2].X,phi_points_dc[i][2].Y);


cdots.x=phi_points[i][1].X;
cdots.y=phi_points[i][1].Y;
cdots.x/=zoomflag;
cdots.y/=zoomflag;
cdots.x+=rc1.left;
cdots.y+=rc1.top;

pDC->SetPixel(cdots,RGB(0,255,0));
pDC->SetPixel(cdots.x,cdots.y+1,RGB(0,255,0));
pDC->SetPixel(cdots.x,cdots.y-1,RGB(0,255,0));
pDC->SetPixel(cdots.x-1,cdots.y-1,RGB(0,255,0));
pDC->SetPixel(cdots.x+1,cdots.y+1,RGB(0,255,0));
pDC->SetPixel(cdots.x+1,cdots.y-1,RGB(0,255,0));
pDC->SetPixel(cdots.x-1,cdots.y+1,RGB(0,255,0));
pDC->SetPixel(cdots.x-1,cdots.y,RGB(0,255,0));
pDC->SetPixel(cdots.x+1,cdots.y,RGB(0,255,0));


cdots.x=phi_points[i][2].X;
cdots.y=phi_points[i][2].Y;
cdots.x/=zoomflag;
cdots.y/=zoomflag;
cdots.x+=rc1.left;
```

```
 cdots.y+=rc1.top;

 pDC->SetPixel(cdots,RGB(0,255,0));
 pDC->SetPixel(cdots.x,cdots.y+1,RGB(0,255,0));
 pDC->SetPixel(cdots.x,cdots.y-1,RGB(0,255,0));
 pDC->SetPixel(cdots.x-1,cdots.y-1,RGB(0,255,0));
 pDC->SetPixel(cdots.x+1,cdots.y+1,RGB(0,255,0));
 pDC->SetPixel(cdots.x+1,cdots.y-1,RGB(0,255,0));
 pDC->SetPixel(cdots.x-1,cdots.y+1,RGB(0,255,0));
 pDC->SetPixel(cdots.x-1,cdots.y,RGB(0,255,0));
 pDC->SetPixel(cdots.x+1,cdots.y,RGB(0,255,0));

 //increment counter

 i++;
 }

}

void CrobthirdView::findPhi1()
{
   int i,j;
   float temp_counter[20];
   for(i=0;i<20;i++) {phi[i]=0;}
   float tempx[20];
   float tempy[20];

   //reuse temp counter
   for(i=0;i<20;i++) {temp_counter[i]=0; tempx[i]=-1;tempy[i]=-1;}


   for(i=1;i<=CF[1].features.num;i++){
     for(j=1;j<=numObjects;j++){
   if(fabs(CF[1].worldpts.z[i]-centres[j].z)<THRESHOLD){
   //this keeps the number of points already added, used to find the average
   temp_counter[j]++;
   float x_av=(CF[1].worldpts.x[i]-centres[j].x)*(CF[1].worldpts.x[i]-centres[j].x);
   float y_av=(CF[1].worldpts.y[i]-centres[j].y)*(CF[1].worldpts.y[i]-centres[j].y);

   if(tempx[j]==-1)
     tempx[j]=x_av;
   else
     tempx[j]=(tempx[j]*temp_counter[j]+x_av)/(temp_counter[j]+1);

   if(tempy[j]==-1)
     tempy[j]=y_av;
   else
     tempy[j]=(tempy[j]*temp_counter[j]+y_av)/(temp_counter[j]+1);

     }}}

   //standard deviation, take sqrt
```

```
    for(j=1;j<=numObjects;j++)
      {
      tempx[j]=sqrt(tempx[j]);
      tempy[j]=sqrt(tempy[j]);
      }

    //now fine phi


    for(j=1;j<=numObjects;j++)
      {
      phi[j]=atan(tempy[j]/tempx[j]);
      }


}
void CrobthirdView::findPhi2()
{
    int i,j;
    float temp_counter[20];
    for(i=0;i<20;i++) {phi[i]=0;}
    float tempx[20];
    float tempy[20];

    //reuse temp counter
    for(i=0;i<20;i++) {temp_counter[i]=0; tempx[i]=-1;tempy[i]=-1;}

    float alpha11,alpha02,alpha20;

    for(j=1;j<=numObjects;j++)
      {
      alpha11=moment(1,1,j);
      alpha20=moment(2,0,j);
      alpha02=moment(0,2,j);
      phi[j]=0.5*(atan2((2*alpha11),(alpha20-alpha02)));
      //if(phi[j]<0) phi[j]+=PI;
      float angle=phi[j]*180/PI;

      //if(phi[j]<0) phi[j]+=PI;
      if(phi[j]>PI) phi[j]-=PI;
      if(phi[j]>(PI/2)) phi[j]-=PI;
      }

}

void CrobthirdView::performfull1()
{
OnReconstructionLoad() ;
OnFeaturesMoravec() ;
}

float CrobthirdView::moment(int m, int n,int object_number)
```

```
{


float tempxy=0;
float tempx,tempy;
int cc=0;
  for(int i=1;i<=CF[1].features.num;i++)
    {
    for(int j=1;j<=CF[1].features.num;j++)
      {
      if(fabs(CF[1].worldpts.z[i]-centres[object_number].z)<THRESHOLD)
        {
        cc++;
        tempx=pow((CF[1].worldpts.x[i]-centres[object_number].x),(float)m);
        tempy=pow((CF[1].worldpts.y[i]-centres[object_number].y),(float)n);
        tempxy+=(tempx*tempy);
        }
      }
    }

  tempxy=tempxy/((float)cc);

  return tempxy;

}

void CrobthirdView::grab1()
{
robot myrob;
myrob.initilize();
float yy=60;
for(int i=1;i<=numObjects;i++){

myrob.moveto2(centres_sorted[i].x,centres_sorted[i].y,centres_sorted[i].z+1,0,(phi[i]*180/PI));
myrob.OPEN();
myrob.moveto2(centres_sorted[i].x,centres_sorted[i].y,centres_sorted[i].z-2,0,(phi[i]*180/PI));
myrob.CLOSE();
myrob.SEND("GF 1");
myrob.moveto2(centres_sorted[i].x,centres_sorted[i].y,centres_sorted[i].z+3,0,(phi[i]*180/PI));
myrob.moveto2(40,yy,centres_sorted[i].z-2,0,0);
myrob.SEND("GF 0");
myrob.OPEN();
myrob.moveto2(40,yy,10,0,0);
yy+=5;
  }
myrob.home2();
myrob.closecomm();
}

void CrobthirdView::OnRobotMoveaside()
{
    robot myrob;
```

```
myrob.initilize();
myrob.home2();
myrob.closecomm();

}

void CrobthirdView::OnRobotGrab1()
{
  robot myrob;
myrob.initilize();
myrob.moveto(centres[1].x,centres[1].y,2,180,54.8000+180);
//myrob.OPEN();
//myrob.CLOSE();
//myrob.SEND("GF 1");
myrob.closecomm();

}

void CrobthirdView::OnRobotOpen()
{
  robot myrob;
myrob.initilize();
myrob.OPEN();
myrob.SEND("GF 0");
myrob.closecomm();
}

void CrobthirdView::OnRobotNest()
{
robot myrob;
myrob.initilize();
myrob.NEST();
myrob.closecomm();
}

void CrobthirdView::OnRobotHome()
{
robot myrob;
myrob.initilize();
  myrob.OPEN();
  myrob.HOME();
myrob.closecomm();
}

void CrobthirdView::OnRobotReset()
{
robot myrob;
myrob.initilize();
myrob.reset();
myrob.closecomm();

}
```

```
void CrobthirdView::OnRobotCalibrationposition1()
{
  robot myrob;
myrob.initilize();
myrob.grabtarget();
myrob.calibrb(4);
myrob.calibrb(1);
myrob.closecomm();

}

void CrobthirdView::OnRobotIntellicam()
{

 HINSTANCE hInst = ShellExecute(0,
                        "open",                 // Operation to perform
                        "C:\\Program Files\\Matrox Imaging\\intellicam\\intelcam.exe",  //
Application name
                        0,        // Additional parameters
                        0,                  // Default directory
                        SW_SHOW );



}

void CrobthirdView::OnRobotPutback()
{
robot myrob;
myrob.initilize();
myrob.putcalibback();
  myrob.closecomm();
}
CPoint CrobthirdView::convertPoint(CPoint pt)
{
CPoint ppt;
ppt.x=0;
ppt.y=0;
if(rc1.PtInRect(pt)==0 && rc2.PtInRect(pt)==0 ) return ppt;

if(rc1.PtInRect(pt))
  {

  ppt.x=pt.x-rc1.left;
  ppt.y=pt.y-rc1.top;

  ppt.x= ppt.x*zoomflag;
  ppt.y= ppt.y*zoomflag;
  return ppt;
  }
```

```
if(rc2.PtInRect(pt))
  {

  ppt.x=pt.x-rc2.left;
  ppt.y=pt.y-rc2.top;

  ppt.x= ppt.x*zoomflag;
  ppt.y= ppt.y*zoomflag;


return ppt;
  }
return NULL;
}

void CrobthirdView::OnCheckShowdisparitygradient()
{
if( disparity_gradient_demo==1) disparity_gradient_demo=0;
else disparity_gradient_demo=1;

disparity_gradient_demo_counter=0;
Invalidate();
}

void CrobthirdView::createDGPoints(CPoint apt)
{
if(rc1.PtInRect(apt)==0 && rc2.PtInRect(apt)==0 ) return;
if(disparity_gradient_demo==0) return;


if(disparity_gradient_demo_counter<4)
  {
  disparity_gradient_demo_counter++;
  if(rc1.PtInRect(apt))
    {
    if(disparity_gradient_demo_counter==1)  pt_left1=apt;
    else if(disparity_gradient_demo_counter==2) pt_left2=apt;
    else {MessageBox("Left Image is Already Full"); disparity_gradient_demo_counter--; return;}
    }

  if(rc2.PtInRect(apt))
    {
    if(disparity_gradient_demo_counter==3)  pt_right1=apt;
    else if(disparity_gradient_demo_counter==4) pt_right2=apt;
    else {MessageBox("Right Image is Already Full"); disparity_gradient_demo_counter--; return;}
    }

  }
else if(disparity_gradient_demo_counter==4)
  {
  m_pointchosen=findClosestNeighbour(apt);
```

```
  disparity_gradient_demo_counter++;
  }
else if(disparity_gradient_demo_counter==5)
  {

  switch(m_pointchosen)
    {
    case 1:
      pt_left1=apt;
      break;
    case 2:
      pt_left2=apt;
      break;
    case 3:
      pt_right1=apt;
      break;
    case 4:
      pt_right2=apt;
      break;
    }

  disparity_gradient_demo_counter=4;//revert back to normal mode

  }
else MessageBox("invalid value for DG demo flags");

Invalidate();

}

int CrobthirdView::findClosestNeighbour(CPoint apt)
{
int d1=findDistance(apt,pt_left1);
int d2=findDistance(apt,pt_left2);
int d3=findDistance(apt,pt_right1);
int d4=findDistance(apt,pt_right2);

int mini=min(min(d1,d2),min(d3,d4));

int pt_num;
    if(mini==d1)
      pt_num=1;

    if(mini==d2)
      pt_num=2;

    if(mini==d3)
      pt_num=3;

    if(mini==d4)
      pt_num=4;
```

```
return pt_num;
}

void CrobthirdView::drawDGdemopts(CDC *pDC)
{
if(disparity_gradient_demo==0 || disparity_gradient_demo_counter==0) return;

if(disparity_gradient_demo_counter>0)
markPoint(pt_left1,pDC,1,4,4);


if(disparity_gradient_demo_counter>1)
markPoint(pt_left2,pDC,1,4,4);


if(disparity_gradient_demo_counter>2)
markPoint(pt_right1,pDC,1,4,4);

if(disparity_gradient_demo_counter>3)
markPoint(pt_right2,pDC,1,4,4);

if(disparity_gradient_demo_counter==5)
  {
if( m_pointchosen==1)
markPoint(pt_left1,pDC,1,4,3);

if( m_pointchosen==2)
markPoint(pt_left2,pDC,1,4,3);
if( m_pointchosen==3)
markPoint(pt_right1,pDC,1,4,3);


if( m_pointchosen==4)
markPoint(pt_right2,pDC,1,4,3);
  }
if(disparity_gradient_demo_counter==3)
  {

for(int i=rc2.top;i<=rc2.bottom;i++)
  for(int j=rc2.left;j<=rc2.right;j++)
    {
    pt_right2.x=j;
    pt_right2.y=i;
    CPoint temppt=convertPoint(pt_right2);
    float
DG=disparityGradient(convertPoint(pt_left1),convertPoint(pt_left2),convertPoint(pt_right1),con
vertPoint(pt_right2));
    if(DG<0.01)
      {
      if(DG<0) MessageBox("whaa");
      float CL=2550-DG*((float)2550);
      pDC->SetPixel(pt_right2,RGB((int)CL,0,0));
```

```
    }
   }
  }


if(disparity_gradient_demo_counter==4)
  {
  CRect rctemp(0,0,70,20);
  float
DG=disparityGradient(convertPoint(pt_left1),convertPoint(pt_left2),convertPoint(pt_right1),con
vertPoint(pt_right2));
  CString b;
  b.Format("%0.4f",DG);
  pDC->DrawText(b.GetBuffer(b.GetLength()),-1,&rctemp,DT_CENTER );
  }


}
void CrobthirdView::OnMatchingHopfieldtest1()
{
CHStereo mystereo(&CF[1],&CF[2],&image[1],&image[2]);
mystereo.findEnergyFromFile("matches2.dat");
Invalidate();

}
void CrobthirdView::OnMatchingMatchhopfield()
{
OnReconstructionLoad() ;
reset_display();
CF[1].moravec(&image[1]);



CHStereo mystereo(&CF[1],&CF[2],&image[1],&image[2],&CFun);
mystereo.correspond();

//CFun.find3D_and_disp(&CF[1],&CF[2]);
//CF[1].calc_statistics();
//findorientation();//find centroid
Invalidate();
CFun.find3D_and_disp(&CF[1],&CF[2]);
CF[1].calc_statistics();
}


void CrobthirdView::OnMatchingHopfieldmatch()
{
  // TODO: Add your command handler code here
CHStereo mystereo(&CF[1],&CF[2],&image[1],&image[2],&CFun);
mystereo.correspond();

Invalidate();
```

```
CFun.find3D_and_disp(&CF[1],&CF[2]);
CF[1].calc_statistics();
}

void CrobthirdView::OnMouseMove(UINT nFlags, CPoint point)
{
CPoint zpoint;
  int gray=0;

 CMainFrame *pMainWnd = (CMainFrame *)AfxGetMainWnd();

  CString s,w;


  if(rc1.PtInRect(point))
  {zpoint.x=point.x-rc1.left;zpoint.y=point.y-
rc1.top;zpoint.y*=zoomflag;zpoint.x*=zoomflag;gray=image[1].getpixel(zpoint.x,zpoint.y);}
  if(rc2.PtInRect(point))
  {zpoint.x=point.x-rc2.left;zpoint.y=point.y-
rc2.top;zpoint.y*=zoomflag;zpoint.x*=zoomflag;gray=image[2].getpixel(zpoint.x,zpoint.y);}


  if(rc1.PtInRect(point) || rc2.PtInRect(point) ){
  s.Format("X=%d Y=%d  Intensity=%d",zpoint.x,zpoint.y,gray);
   pMainWnd->m_wndStatusBar.SetPaneText(0,s);
  w.Format("Gray Level=%d",gray);
  pMainWnd->m_wndStatusBar.SetPaneText(2,w);

}
CFormView::OnMouseMove(nFlags, point);
}

void CrobthirdView::OnGraphGraph1()
{
CHFunctiongraph myf;
if(myf.DoModal()==IDOK) ;
}

void CrobthirdView::OnCheckCheckhrbitmap()
{
unsigned char* b=new unsigned char[200*200 + 10];

for(int i=1;i<=200;i++)
  {
  for(int j=1;j<=100;j++)
    {
    b[((i-1)*100)+j]=250;
    }
  }

image[1].open(b,100,200);
image[2].open(b,100,200);
```

```
fixarea();
imagesloaded=1;
Invalidate();
delete [] b;

}

void CrobthirdView::OnMatchingDensematch1()
{OnReconstructionLoad() ;
reset_display();
CF[1].canny(&image[1]);
CF[2].canny(&image[2]);
  // TODO: Add your command handler code here
  CHopdense dense(&image[1],&image[2],&CF[1],&CF[2]);
  dense.dense_match_1(1);
CFun.find3D_and_disp(&CF[1],&CF[2]);
CF[1].calc_statistics();
Invalidate();
}

void CrobthirdView::OnCheckIncreasecontrastofimage1()
{

float temp1=0;
float temp2=(float)2;

for(int i=1;i<=image[1].height;i++)
  {
  for(int j=1;j<=image[1].width;j++)
    {

    temp1=image[1].getpixel(j-1,i-1);
    temp1*=temp2;

    if(temp1>255) temp1=255;
    if(temp1<0) temp1=0;


    image[1].setpixel(j-1,i-1,(int)temp1);
    }
  }
Invalidate();

}

void CrobthirdView::OnMatchingDensematchcorrelation()
{
  OnReconstructionLoad() ;
reset_display();

  // TODO: Add your command handler code here
  CHopdense dense(&image[1],&image[2],&CF[1],&CF[2]);
```

```
  dense.dense_match_1(2);
    CF[1].edge_display_flag=3;
    CF[2].edge_display_flag=3;
Invalidate();

}

void CrobthirdView::OnCheckEnablelinematching()
{
  // TODO: Add your command handler code here
if(enable_line_match==0)
  enable_line_match=1;
else
enable_line_match=0;
  }
void CrobthirdView::OnCheckForcereloadimage1()
{
  // TODO: Add your command handler code here
image[1].open("..\\images\\right.bmp");

  fixarea();
imagesloaded=1;


Invalidate();
}
void CrobthirdView::OnCheckForcereloadimage2()
{
//image[0].open("images\\truedisp.bmp");

image[2].open("..\\images\\left.bmp");
fixarea();
imagesloaded=1;


Invalidate();

}

void CrobthirdView::OnFeaturesIncreasesensitivity()
{
  if(sensitivity>3) sensitivity=sensitivity-2;
  reset_display();
float s=0.8;
float temp=sensitivity;
s=s+((temp-50)/100);
CF[1].canny(&image[1],s);
CF[2].canny(&image[2],s);

  Invalidate();

}
```

```
void CrobthirdView::OnFeaturesDecreasesensitivity()
{

    if(sensitivity<130) sensitivity=sensitivity+2;
    reset_display();


float s=0.8;
float temp=sensitivity;
s=s+((temp-50)/100);
CF[1].canny(&image[1],s);
CF[2].canny(&image[2],s);

  Invalidate();
}

void CrobthirdView::OnFeaturesResetsensitivity()
{
sensitivity=50;
reset_display();
float s=0.8;
float temp=sensitivity;
s=s+((temp-50)/100);
CF[1].canny(&image[1],s);
CF[2].canny(&image[2],s);

  Invalidate();

}

void CrobthirdView::OnMatchingFulldensematchingcorrelation()
{
  OnReconstructionLoad() ;
reset_display();

  // TODO: Add your command handler code here
  CHopdense dense(&image[1],&image[2],&CF[1],&CF[2]);
  dense.dense_match_1(3);

Invalidate();
}

void CrobthirdView::OnCheckShowgroundtruth()
{
  image[2].open("..\\images\\truedisp.bmp");
fixarea();
Invalidate();
}
```

// robthirdView.h : interface of the CrobthirdView class

```
//
/////////////////////////////////////////////////////////////////////////////
//

#if
!defined(AFX_robthirdVIEW_H__68D41DBF_03F0_47B2_A86E_6A4DD66F7285__INCLU
DED_)
#define
AFX_robthirdVIEW_H__68D41DBF_03F0_47B2_A86E_6A4DD66F7285__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
#include "HRBitmap.h"
#include "Features.h"
#include "Fundamental2.h"


class CrobthirdView : public CFormView
{
protected: // create from serialization only
  CrobthirdView();
  DECLARE_DYNCREATE(CrobthirdView)

public:
  //{{AFX_DATA(CrobthirdView)
  enum{ IDD = IDD_robthird_FORM };
    // NOTE: the ClassWizard will add data members here
  //}}AFX_DATA

// Attributes
public:
  CrobthirdDoc* GetDocument();

// Operations
public:

// Overrides
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CrobthirdView)
  public:
  virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
  virtual BOOL PreTranslateMessage(MSG* pMsg);
  protected:
  virtual void DoDataExchange(CDataExchange* pDX);   // DDX/DDV support
  virtual void OnInitialUpdate(); // called first time after construct
  virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
  virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
  virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
  virtual void OnPrint(CDC* pDC, CPrintInfo* pInfo);
  virtual void OnDraw(CDC* pDC);
  //}}AFX_VIRTUAL
```

```
// Implementation
public:
  int enable_line_match;
  void drawDGdemopts(CDC* pDC);
  int findClosestNeighbour(CPoint apt);
  void createDGPoints(CPoint apt);
  //void grab1(void);
  float moment(int m,int n,int object_number);
  void performfull1(void);
  void findPhi1(void);
  void findPhi2(void);
  void drawprojection(CDC* pDC);
  void addCluster(float value,int K);
  void findorientation(void);
  void reset_display(void);

  void clean_grid(float** pgrid,int m,int n);
  int matchall(void);
  int matchall_1(void);


  CPoint convertPoint(CPoint pt);
  CPoint fixpoint(CPoint p,CRect rc, float zoom);
  void load_F_matrix(void);
  void matchlines(CPoint cp);
  void writetitle(CDC* pDC);
  void fixarea(void);
  void changebackdir();
  void GRAB();
  void cleandirectory();
  int modeflag;
  wpoints2 centres_sorted[20];
  int imagesloaded;



  HRBitmap image[3];//left and right image
  CRect rc1,rc2,rc3;
  CFeatures CF[3];
  float phi[20];
  tpoint phi_points[20][3];
  tpoint phi_points_dc[20][3];
  cluster surfaces[40];
  wpoints2 centres[20];
  tpoint proj_centres[20];
  CPoint* matches;
  int sensitivity;//feature extractor's inittial sensitivity
  int match_array_loaded;
  CPoint pt_left1,pt_left2,pt_right1,pt_right2;
  int m_pointchosen;//for DG dispaly
  int disparity_gradient_demo;
```

```
  int disparity_gradient_demo_counter;
  int numObjects;
    int showprojection;
    float zoomflag;
    int drawmatches;
    int F_matrix_loaded;
    float background;
    int ipshow_value;
    int match_lines;
    CFundamental2 CFun;
  virtual ~CrobthirdView();
#ifdef _DEBUG
  virtual void AssertValid() const;
  virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
  //{{AFX_MSG(CrobthirdView)
  afx_msg void OnModeFile();
  afx_msg void OnModeCamera();
  afx_msg void OnReconstructionLoad();
  afx_msg void OnZoomZoomout();
  afx_msg void OnZoomZomin();
  afx_msg void OnZoomReset();
  afx_msg void OnFeaturesSobel();
  afx_msg void OnFeaturesSusancorners();
  afx_msg void OnFeaturesSusanedges();
  afx_msg void OnFeaturesMoravec();
  afx_msg void OnFeaturesCanny();
  afx_msg void OnFeaturesReset();
  afx_msg void OnFeaturesRemovebackground();
  afx_msg void OnFeaturesChangedisplaystyle();
  afx_msg void OnMatchingMatchlines();
  afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
  afx_msg void OnMatchingMatchall();
  afx_msg void OnFeaturesReducefeatures();
  afx_msg void OnVScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar);
  afx_msg void OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar);
  afx_msg void OnMatchingMatchall2();
  afx_msg void OnCheckCheck();
  afx_msg void show_differentvalue();
  afx_msg void unshow_value();
  afx_msg void OnPrintdataPrintfeatures();
  afx_msg void OnShowpointsShowdispand3d();
  afx_msg void OnShowpointsNewwindowforstats();
  afx_msg void OnShowpointsShowprojection();
  afx_msg void OnRobotMoveaside();
  afx_msg void OnRobotGrab1();
  afx_msg void OnRobotOpen();
```

```
 afx_msg void OnRobotNest();
 afx_msg void OnRobotHome();
 afx_msg void OnRobotReset();
 afx_msg void OnRobotCalibrationposition1();
 afx_msg void OnRobotIntellicam();
 afx_msg void OnRobotPutback();
 afx_msg void grab1();
 afx_msg void OnMatchingMatchhopfield();
 afx_msg void OnMatchingHopfieldtest1();
 afx_msg void OnCheckShowdisparitygradient();
 afx_msg void OnMatchingHopfieldmatch();
 afx_msg void OnMouseMove(UINT nFlags, CPoint point);
 afx_msg void OnGraphGraph1();
 afx_msg void OnCheckCheckhrbitmap();
 afx_msg void OnMatchingDensematch1();
 afx_msg void OnCheckIncreasecontrastofimage1();
 afx_msg void OnMatchingDensematchcorrelation();
 afx_msg void OnCheckEnablelinematching();
 afx_msg void OnCheckForcereloadimage1();
 afx_msg void OnCheckForcereloadimage2();
 afx_msg void OnFeaturesIncreasesensitivity();
 afx_msg void OnFeaturesDecreasesensitivity();
 afx_msg void OnFeaturesResetsensitivity();
 afx_msg void OnMatchingFulldensematchingcorrelation();
 afx_msg void OnCheckShowgroundtruth();
 //}}AFX_MSG
 DECLARE_MESSAGE_MAP()
};

#ifndef _DEBUG  // debug version in robthirdView.cpp
inline CrobthirdDoc* CrobthirdView::GetDocument()
   { return (CrobthirdDoc*)m_pDocument; }
#endif

/////////////////////////////////////////////////////////////////////////////
//

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the previous line.

#endif //
!defined(AFX_robthirdVIEW_H__68D41DBF_03F0_47B2_A86E_6A4DD66F7285__INCLU
DED_)
```

// HStereo.cpp: implementation of the CHStereo class.
//implementation of matching

```
/////////////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "robthird.h"
```

```
#include "Hopfield.h"
#include "HStereo.h"
#include "HEnergy.h"
#include "MATX.h"
#include "HSDialog.h"
#include <fstream.h>

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif

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

CHStereo::CHStereo()
{
  }


CHStereo::CHStereo(CFeatures* cf1,CFeatures* cf2,HRBitmap* cb1,HRBitmap*
cb2,CFundamental2* CFun)
  {

init(cf1,cf2,cb1,cb2,CFun);
  }


CHStereo::CHStereo(CFeatures* cf1,CFeatures* cf2,HRBitmap* cb1,HRBitmap* cb2)
{
init(cf1,cf2,cb1,cb2);
}

CHStereo::~CHStereo()
{
}

void CHStereo::correspond()
{

//main function that calls most other functions in this class
//write the features to file
m_leftfeature->printfeatures("leftfeatures.txt");
//form the 2D hopfield Neural Network
float networkwidth;//determine if its is a regular neurla network or the new formulation where
the height represents the disparity
float networkheight;
if(m_operationMode!=3){

m_rightfeature->moravec(m_rightimage);
```

```
networkwidth=m_rightfeature->features.num;
networkheight=m_leftfeature->features.num;
  }
else {
networkwidth=m_leftfeature->features.num;
networkheight=m_maxdisp+1;//if maximum disparity is 10, we need 11 disparity levels, the extra
is for zero
  }


///declare the neural network
CHopfield
mHopfield(networkheight,networkwidth,ON,OFF,m_operationMode,m_activationalpha,m_initty
pe,m_initvalue,m_gaussiandeviation,m_initialneuronvalue,m_updatingmode,0);//height is left
features, or each row is one left feature, convention

//m_leftimage->setallto(150);
//m_rightimage->setallto(150);


if(m_inittype==6)
initCorrelate(&mHopfield);//for initialization in case of correlation init

mHopfield.WriteActivations("initialactivations.txt");
//now set the weights
if(m_operationMode!=3)
setBiasAll(&mHopfield,(float)2);//every neuron has a bias of two
else
setBiasAll(&mHopfield,(float)1);//my formulation requires uniqueness only in one column not in
rows


if( m_operationMode!=2)//dont update if its simple correlation
setWeights(&mHopfield);
//zeroWeights(&mHopfield);
//mHopfield.alignWeights();
//artificial weights
//setAllWeights(&mHopfield,-1);
//mHopfield.setConnectionValue(2,1,3,2,1);
//mHopfield.setConnectionValue(2,2,3,1,1);

if(m_writeweights==TRUE)  mHopfield.writeWeightstoFile("weights.txt");
if(m_writeactivations==TRUE) mHopfield.WriteActivations("initial-activations.txt");
//now update the grid
if( m_operationMode!=2)//dont update if its simple correlation
doUpdateNetwork(&mHopfield);


if(m_writeactivations==TRUE) mHopfield.WriteActivations("activations.txt");
//now prune network
int temp;
```

```
if(m_operationMode!=3)
temp=pruneNetwork_3(&mHopfield);
else
temp=pruneNetwork_4(&mHopfield);

if(m_writeactivations==TRUE) mHopfield.WriteActivations("activations-pruned.txt");
//temp=pruneNetwork_1(&mHopfield);unnessary

//now report excess matches number,
if(m_multiplematches==TRUE)
  {
  CString b;
  b.Format("Number of multiple matches is %d",temp);
  AfxMessageBox(b);
  }


PointMatch* pt_match=formMatches(&mHopfield);

///write matches to text file
FILE* tfile;
tfile=fopen("matches.txt","wt");
  for(int i=1;i<=m_leftfeature->features.num;i++){
    fprintf(tfile,"%d   ----------
%d\n",pt_match[i].LeftFeatureNumber,pt_match[i].RightFeatureNumber);
    }
fclose(tfile);



//now sort the features
sortFeatures(pt_match);

//release the memory for the matches array
delete [] pt_match;

if(m_showactivationfunction==TRUE) drawFunction(&mHopfield,0);
if(m_compatibilityshow==TRUE) drawFunction(&mHopfield,1);
//once the features are sorted #d can be found

//create match list and sort features as before with the corrleation thing

//find 3D coordinatess
//TO do afterwards

//add more constraints and modify the objective function add more terms
}

void CHStereo::setWeights(CHopfield *hopfield)
{
```

```
//this function gets a pointer to a hopfield neural network
//and sets its weights
int i,j,p,q;


//go through the network and set the weights


//go throug all connections and set the weights
for( i=1;i<=hopfield->m_height;i++)
  {
  for( j=1;j<=hopfield->m_width;j++)
    {
    for( p=1;p<=hopfield->m_height;p++)
      {
      for( q=1;q<=hopfield->m_width;q++)
        {
        if(i!=p || j!=q)//no self feedback
          {
          if(hopfield->m_neuronGrid[i][j].m_connection[p][q].done==0 && hopfield-
>m_neuronGrid[p][q].m_connection[i][j].done==0)
            {

            hopfield-
>setConnectionValue(i,j,p,q,scaleweight(i,j,p,q,calculateWeight(i,j,p,q))+m_offset);
            }
          }
        }
      }
    }
  }



}

void CHStereo::doUpdateNetwork(CHopfield *hopfield)
  {

  if(m_updatingmode==0)
    doUpdateNetwork1(hopfield);
  else
    doUpdateNetwork2(hopfield);


  }

void CHStereo::doUpdateNetwork1(CHopfield *hopfield)
{
//this function takes the pointer to the neural network and updates
```

```
//in this updating method we update the network until the energy doesnt change for 200
consecutive iterations
//if the number of iterations passes 20000  it will stop
int numTimesEnergyIncreased=0;
float energy_present=0;
float energy_past=-1;
int numberIterations=0;
float numberConsecutive=0;

float *energyValues=new float[m_maxiterations+2];
//loop until networks settles down
//the conditions are 1- loop is still hasnt settled down
//2- the max numiterations hasnt been reached


while(numberConsecutive<m_stableiterations && numberIterations<m_maxiterations)
  {


if(numberIterations==271)
int z=2;
  numberIterations++;//increase te counter
  energy_past=energy_present;
  //now is the actual updating of the network
  energyValues[numberIterations]=energy_present=hopfield->updateNetwork(1);//update once
every time
  //energyValues[numberIterations]=energy_present= hopfield->findEnergyDiscrete();
  //increase the counter denoting how many times the network has been stale (for how many
iterations)


  if(energy_present>(energy_past+0.1) && numberIterations>1){
    numTimesEnergyIncreased++;
    if(m_tellenergyincrease==TRUE){
    CString b;
    b.Format("%d",numberIterations);
    AfxMessageBox(b);
    }}

  if(fabs(energy_present-energy_past)<0.001)
    numberConsecutive++;
  else
    if(numberConsecutive>1)
    numberConsecutive=0;
  }



//creating the dialog that will show the energy graph
writevector(energyValues,numberIterations,"energy.txt");
if(m_showenergywindow==TRUE){
```

```
CHEnergy myen;
myen.m_energy=energy_past;
myen.m_iterations=numberIterations;
myen.setData(numberIterations,energyValues);
if(myen.DoModal()==IDOK);}
delete [] energyValues;

  // CString b;
  // b.Format("%d",numTimesEnergyIncreased);
  // AfxMessageBox(b);
}
void CHStereo::doUpdateNetwork2(CHopfield *hopfield)
{
//winner takes all
int numTimesEnergyIncreased=0;
float energy_present=0;
float energy_past=-1;
int numberIterations=0;
float numberConsecutive=0;

float *energyValues=new float[m_maxiterations+2];

while(numberConsecutive<m_stableiterations && numberIterations<m_maxiterations)
  {


if(numberIterations==271)
int z=2;
  numberIterations++;//increase te counter
  energy_past=energy_present;
  //now is the actual updating of the network
  energyValues[numberIterations]=energy_present=hopfield-
>updateNetwork((numberIterations%hopfield->m_width)+1);//update once every time
  //energyValues[numberIterations]=energy_present= hopfield->findEnergyDiscrete();
  //increase the counter denoting how many times the network has been stale (for how many
iterations)


  if(energy_present>(energy_past+0.1) && numberIterations>1){
    numTimesEnergyIncreased++;
    if(m_tellenergyincrease==TRUE){
    CString b;
    b.Format("%d",numberIterations);
    AfxMessageBox(b);
    }}

  if(fabs(energy_present-energy_past)<0.001)
    numberConsecutive++;
  else
    if(numberConsecutive>1)
    numberConsecutive=0;
```

```
 }


//creating the dialog that will show the energy graph
if(m_showenergywindow==TRUE){
CHEnergy myen;
myen.m_energy=energy_past;
myen.m_iterations=numberIterations;
myen.setData(numberIterations,energyValues);
if(myen.DoModal()==IDOK);}
delete [] energyValues;

}

float CHStereo::calculateWeight(int i, int j, int p, int q)
{

if(i==10 && j==1 && p==25 && q==2)
int hht=0;

if(m_objectivefunction== 0) return calculateWeight_1(i, j,  p,  q);
if(m_objectivefunction== 1) return calculateWeight_2(i, j,  p,  q);
if(m_objectivefunction== 2) return calculateWeight_3(i, j,  p,  q);

}
float CHStereo::calculateWeight(int i, int j, int p, int q,float wvalue)
{
return (wvalue-findMiu(i,p)-findMiu(j,q));
  }

float CHStereo::findDisparity(float xl, float yl, float xr, float yr)
{
//this finds the distance between two points, thats it

//this function simply finds the disparity of a match, not sure if this is the way to do it for
nonplanar cameras
//in rectified images i would simply use xr-xl

//this is used for both disparity AND distance, is disparith same as distance
float xtemp=(xl-xr)*(xl-xr);
float ytemp=(yl-yr)*(yl-yr);

float dtemp=sqrt(xtemp+ytemp);

return dtemp;

}
void CHStereo::setBiasAll(CHopfield *hopfield, float value)
{
//this function sets a bias value for all neurons in a given network
int i,j;
```

```
//go through the network and set biases
for( i=1;i<=hopfield->m_height;i++)
  {
  for( j=1;j<=hopfield->m_width;j++)
    {

    hopfield->setBiasValue(i,j,value);

    }
  }

}
int CHStereo::pruneNetwork_1(CHopfield *hopfield)
{
//this is the first function to eliminate multiple matches, it will just choose the first candidate
int i,j;
int counter=0;
int tempflag;

//eliminate & count float matches horizontally
for( i=1;i<=hopfield->m_height;i++)
  {
  tempflag=0;
  for( j=1;j<=hopfield->m_width;j++)
    {
    if(hopfield->m_neuronGrid[i][j].activation==ON)
      {
      tempflag++;

      if(tempflag>1) hopfield->m_neuronGrid[i][j].activation=OFF;//turn off excess matches
      }

    }

  counter+=tempflag-1;
  }

//now do the same thing vertically
for( j=1;j<=hopfield->m_width;j++)
  {
  tempflag=0;
  for( i=1;i<=hopfield->m_height;i++)
    {
    if(hopfield->m_neuronGrid[i][j].activation==ON)
      {
      tempflag++;

      if(tempflag>1) hopfield->m_neuronGrid[i][j].activation=OFF;//turn off excess matches
      }

    }
```

```
 counter+=tempflag-1;//add thre number of excess matches
 }


return counter;

}

int CHStereo::pruneNetwork_2(CHopfield *hopfield)
{
//this works for parallel cameras with y disoparity =0, a better way to do this would
//be to measure the average y disparity and check against that
int i,j;
int counter=0;
int tempflag;
float average_y_disparity=0;
float limit=0;

//eliminate & count float matches horizontally
for( i=1;i<=hopfield->m_height;i++)
 {
 tempflag=0;
 for( j=1;j<=hopfield->m_width;j++)
   {
   if(hopfield->m_neuronGrid[i][j].activation>0.95)
     {
     tempflag++;

     if((fabs(m_leftfeature->features.y[i]-m_rightfeature->features.y[j]))>limit) hopfield-
>m_neuronGrid[i][j].activation=OFF;//turn off excess matches
     }
   }

 counter+=tempflag-1;//add thre number of excess matches
 }

return counter;

}
PointMatch* CHStereo::formMatches(CHopfield *hopfield)
{
if(m_operationMode!=3)
return formMatchesFvsF(hopfield);
else
return formMatchesFvsDS(hopfield);

}
PointMatch* CHStereo::formMatchesFvsF(CHopfield *hopfield)
{
int i,j;
PointMatch* matches=new PointMatch[max(hopfield->m_height,hopfield->m_width)+2];
```

```
for(i=1;i<=m_leftfeature->features.num;i++)
  {
  matches[i].LeftFeatureNumber=i;matches[i].RightFeatureNumber=-1;
  for(j=1;j<=m_rightfeature->features.num;j++)
    {
    if(hopfield->m_neuronGrid[i][j].activation==ON) matches[i].RightFeatureNumber=j;
    }
  }
return matches;
}
PointMatch* CHStereo::formMatchesFvsDS(CHopfield *hopfield)
{
int i,j;
PointMatch* matches=new PointMatch[hopfield->m_width+2];


for(j=1;j<=hopfield->m_width;j++)
  {
  matches[j].LeftFeatureNumber=i;
  matches[j].RightFeatureNumber=-1;
  for(i=1;i<=hopfield->m_height;i++)
    {
    if(hopfield->m_neuronGrid[i][j].activation==ON) matches[j].RightFeatureNumber=i-1;
    }
  }
return matches;
}

float CHStereo::findDeltad(int i, int j, int p, int q)
{
float disparity1,disparity2;
disparity1=findDisparity(getLeftImagePoint(i,j).x,getLeftImagePoint(i,j).y,getRightImagePoint(i,j).
x,getRightImagePoint(i,j).y);
disparity2=findDisparity(getLeftImagePoint(p,q).x,getLeftImagePoint(p,q).y,getRightImagePoint(
p,q).x,getRightImagePoint(p,q).y);
return (disparity1-disparity2);
}

float CHStereo::findDeltaD(int i, int j, int p, int q)
{
float distance1,distance2;
distance1=findDisparity(getLeftImagePoint(i,j).x,getLeftImagePoint(i,j).y,getLeftImagePoint(p,q).x
,getLeftImagePoint(p,q).y);
distance2=findDisparity(getRightImagePoint(i,j).x,getRightImagePoint(i,j).y,getRightImagePoint(p
,q).x,getRightImagePoint(p,q).y);
return (distance1-distance2);
}

float CHStereo::compatibility(float X)
{
if(m_compatibilitytype==0) return compatibility_1(X);
```

```
if(m_compatibilitytype==1) return compatibility_2(X);
}

float CHStereo::compatibility_1(float X)
{
float temp1=m_omega*(X-m_teta);
temp1=1+(exp(temp1));
temp1=((float)2.00)/temp1;
return (temp1-1);
}

float CHStereo::compatibility_2(float X)
{
float temp1=(-X/m_T);
temp1=(exp(temp1));
temp1=((float)2.00)*temp1;
return (temp1-1);
}

float CHStereo::findMiu(int i, int j)
{
float temp;
if(i==j) temp=1;
else temp=0;

return temp;
}

void CHStereo::sortFeatures(PointMatch *matches)
{
if(m_operationMode!=3)
sortFeaturesFvsF(matches);
else
sortFeaturesFvsDS(matches);


}
void CHStereo::sortFeaturesFvsDS(PointMatch *matches)
{
m_rightfeature->allocate(m_leftfeature->features.num,'f');//allocate space for right image features
int i;
//since disparity versus features was used we have to form completely new features for right
image
for(i=1;i<=m_leftfeature->features.num;i++)
  {

    m_rightfeature->features.x[i]=matches[i].RightFeatureNumber+m_leftfeature->features.x[i];
    m_rightfeature->features.y[i]=m_leftfeature->features.y[i];

  }
}
void CHStereo::sortFeaturesFvsF(PointMatch *matches)
```

```
{
int i;
//temporary holders for the sort algorithm
 ipoint* lefttemp =new  ipoint[ m_leftfeature->features.num+2];
 ipoint* righttemp=new  ipoint[m_rightfeature->features.num+2];
int cc1=1;//counter for the following loop
//remove any points in the original if no correspondence was found

  for(i=1;i<=m_leftfeature->features.num;i++)
    {
   if(matches[i].RightFeatureNumber!=-1)
     {
     lefttemp[cc1].x=m_leftfeature->features.x[matches[i].LeftFeatureNumber];
     lefttemp[cc1].y=m_leftfeature->features.y[matches[i].LeftFeatureNumber];

     righttemp[cc1].x=m_rightfeature->features.x[matches[i].RightFeatureNumber];
     righttemp[cc1].y=m_rightfeature->features.y[matches[i].RightFeatureNumber];
     cc1++;
     }
    }

  //resize the feature vectors
   m_leftfeature->allocate(cc1-1,'f');
  m_rightfeature->allocate(cc1-1,'f');


  for(i=1;i<cc1;i++)
    {
     m_leftfeature->features.x[i]=lefttemp[i].x;
     m_leftfeature->features.y[i]=lefttemp[i].y;

     m_rightfeature->features.x[i]=righttemp[i].x;
     m_rightfeature->features.y[i]=righttemp[i].y;
    }


delete [] lefttemp;
delete [] righttemp;

}

void CHStereo::setAllWeights(CHopfield* hopfield,float value)
{

//this function gets a pointer to a hopfield neural network

int i,j,p,q;


//go through the network and set the weights
```

```
//go throug all connections and set the weights
for( i=1;i<=hopfield->m_height;i++)
  {
  for( j=1;j<=hopfield->m_width;j++)
    {
    for( p=1;p<=hopfield->m_height;p++)
      {
      for( q=1;q<=hopfield->m_width;q++)
        {
        if(i!=p || j!=q)//no self feedback
          {
          if(hopfield->m_neuronGrid[i][j].m_connection[p][q].done==0 && hopfield-
>m_neuronGrid[p][q].m_connection[i][j].done==0)          {
            hopfield->setConnectionValue(i,j,p,q,calculateWeight(i,j,p,q,value-1));
            }
          }
        }
      }
    }
  }

}


int CHStereo::pruneNetwork_3(CHopfield *hopfield)
{
//this applies only to the ontinues hopfield neural network and it should
//not be applied to continues, it prunes the network by finding the maximum in the row and
columns

//this is the first function to eliminate multiple matches, it will just choose the first candidate
int i,j;
int counter=0;
int tempflag;

//eliminate & count float matches horizontally
float max;
int jindex=0;
int iindex=0;

float averageactivation=0.1;
for( i=1;i<=hopfield->m_height;i++)
  for( j=1;j<=hopfield->m_width;j++)
    averageactivation+=hopfield->m_neuronGrid[i][j].activation;

averageactivation=0;


for( i=1;i<=hopfield->m_height;i++)
  {
  tempflag=0;
  max=-10000;
```

```
 jindex=-1;
 for( j=1;j<=hopfield->m_width;j++)
   {
   if(hopfield->m_neuronGrid[i][j].activation>max && hopfield-
>m_neuronGrid[i][j].activation>averageactivation)
     {
     tempflag++;
     max=hopfield->m_neuronGrid[i][j].activation;
     jindex=j;
     }
   hopfield->m_neuronGrid[i][j].activation=OFF;//set all neurons in a row to zero then set the
max to the max value

   }
 if(jindex!=-1) hopfield->m_neuronGrid[i][jindex].activation=max;
 counter+=tempflag-1;//add the number of excess matches
 }

//now do the same thing vertically
for( j=1;j<=hopfield->m_width;j++)
 {
 tempflag=0;
 max=-10000;
 iindex=-1;
 for( i=1;i<=hopfield->m_height;i++)
   {
   if(hopfield->m_neuronGrid[i][j].activation>max && hopfield-
>m_neuronGrid[i][j].activation>averageactivation)
     {
     tempflag++;
     max=hopfield->m_neuronGrid[i][j].activation;
     iindex=i;
     }
   hopfield->m_neuronGrid[i][j].activation=OFF;//set all neurons in a row to zero then set the
max to the max value
   }

 if(iindex!=-1) hopfield->m_neuronGrid[iindex][j].activation=ON;//now set the match to one to
indicate match
 counter+=tempflag-1;//add thre number of excess matches
 }


return counter;


}
int CHStereo::pruneNetwork_4(CHopfield *hopfield)
{
//this pruning is for the feature vs desparity formulation
int i,j;
int counter=0;
```

```
int tempflag;

//eliminate & count float matches horizontally
float max;
int iindex=0;

for( j=1;j<=hopfield->m_width;j++)
  {
  tempflag=0;
  max=-10000;
  iindex=-1;
for( i=1;i<=hopfield->m_height;i++)
    {
    if(hopfield->m_neuronGrid[i][j].activation>max )
      {
      max=hopfield->m_neuronGrid[i][j].activation;
      iindex=i;
      }
    hopfield->m_neuronGrid[i][j].activation=OFF;//set all neurons in a column to zero then set
the max to the max value

    }
  if(iindex!=-1) hopfield->m_neuronGrid[iindex][j].activation=ON;

  }

return 0;


}
float CHStereo::findEnergyFromFile(char *filename)
{
ifstream matchfile (filename);

if(!matchfile) {
AfxMessageBox("Cannot open match file.");
return 1;
}


CHopfield mtHopfield(m_leftfeature->features.num,m_rightfeature-
>features.num,ON,OFF,m_operationMode,m_activationalpha,m_inittype,m_initvalue,m_gaussia
ndeviation,m_initialneuronvalue,m_updatingmode,0);//height is left features, or each row is one
left feature, convention
PointMatch* matches=new PointMatch[max(mtHopfield.m_height,mtHopfield.m_width)+2];

int nummatches=1;
while(matchfile)
  {
  matchfile>>matches[nummatches].LeftFeatureNumber;
  matchfile>>matches[nummatches].RightFeatureNumber;
  nummatches++;
```

```
  }
nummatches--;
matchfile.clear();
matchfile.close();
setBiasAll(&mtHopfield,(float)2);//every neuron has a bias of two
setWeights(&mtHopfield);



//now set the activation values to the proper ones



//go throug all connections and set the weights
for( int i=1;i<=mtHopfield.m_height;i++)
  {
  for(int j=1;j<=mtHopfield.m_width;j++)
    {
    mtHopfield.m_neuronGrid[i][j].activation=OFF;
    }
  }
for(i=1;i<nummatches;i++)
  {

  mtHopfield.m_neuronGrid[matches[i].LeftFeatureNumber][matches[i].RightFeatureNumber].ac
tivation=ON;

  }

if(m_writeactivations==TRUE) mtHopfield.WriteActivations("beforeupdating.txt");
doUpdateNetwork(&mtHopfield);
if(m_writeactivations==TRUE) mtHopfield.WriteActivations("afterupdating.txt");
pruneNetwork_3(&mtHopfield);
if(m_writeactivations==TRUE) mtHopfield.WriteActivations("afterpruning.txt");


PointMatch* pt_match=formMatches(&mtHopfield);

//now sort the features
sortFeatures(pt_match);

delete [] pt_match;
delete [] matches;

return 0;
}

void CHStereo::drawFunction(CHopfield *hopfield,int functiontype)
{
int size=200;
float *datay=new float[205];
float *datax=new float[205];
```

```
for(int i=-100;i<=100;i++)
  {

  if(functiontype==0)
    {
    datay[i+100]=hopfield->m_neuronGrid[1][1].activationFunction(i);
    datax[i+100]=i;
    }
  if(functiontype==1)
    {
    datay[i+100]=compatibility(i+100);
    datax[i+100]=i+100;
    }

  }

  if(functiontype==1) size=70;
//creating the dialog that will show the energy graph

CHEnergy myen;
myen.m_energy=m_activationalpha;
myen.m_iterations=m_operationMode;
myen.setData(size,datay,datax);
if(myen.DoModal()==IDOK);
delete [] datax;
delete [] datay;
}

float CHStereo::calculateWeight_1(int i, int j, int p, int q)
{
float W1= 0.4;
float W2=0.6;

//find the value of the objective function variables
float delta_d=fabs(findDeltad(i,j,p,q));
float delta_D=fabs(findDeltaD(i,j,p,q));
float C=compatibility((W1*delta_d)+(W2*delta_D));//C is an intermediate variable

return (C+generalMiu(i,j,p,q));//feature versus feature formulation
}

float CHStereo::calculateWeight_2(int i, int j, int p, int q)
{
float C=calculateDisparityGradient(i,j,p,q);//C is an intermediate variable
return C;
}

float CHStereo::calculateWeight_3(int i,int j,int p,int q)
  {
//this is the third formation of the objective function that uses correlation and distance from the
epipolar line
//the compatibility is the correlation of the first two points plus the corrleationof the second two
```

```
points,, averaged
float cr1=m_cfun-
>ZNCC_2(m_leftimage,m_rightimage,7,getLeftImagePoint(i,j).x,getLeftImagePoint(i,j).y,getRight
ImagePoint(i,j).x,getRightImagePoint(i,j).y);
float cr2=m_cfun-
>ZNCC_2(m_leftimage,m_rightimage,7,getLeftImagePoint(p,q).x,getLeftImagePoint(p,q).y,getRi
ghtImagePoint(p,q).x,getRightImagePoint(p,q).y);
//float C=compatibility(calculateDisparityGradient(i,j,p,q));//C is an intermediate variable
float C=(cr1+cr2+2)/((float)4);

return (C+generalMiu(i,j,p,q));//feature versus feature formulation
}

float CHStereo::cyclopeanSeperation(int i, int j, int p, int q)
{
//the cyclopean location of first point (i,j)
CPoint leftimagepoint1,rightimagepoint1,leftimagepoint2,rightimagepoint2;

leftimagepoint1=getLeftImagePoint(i,j);
rightimagepoint1=getRightImagePoint(i,j);
leftimagepoint2=getLeftImagePoint(p,q);
rightimagepoint2=getRightImagePoint(p,q);
float vector1x=leftimagepoint1.x+rightimagepoint1.x;
float vector1y=leftimagepoint1.y+rightimagepoint1.y;
//average
vector1x/=((float)2);
vector1y/=((float)2);


//the cyclopean location of second point (i,j)
float vector2x=leftimagepoint2.x+rightimagepoint2.x;
float vector2y=leftimagepoint2.y+rightimagepoint2.y;
//average
vector2x/=((float)2);
vector2y/=((float)2);


//distance between the two cyclopean points
float vectorx=vector2x-vector1x;
float vectory=vector2y-vector1y;


//vector norm
float norm=sqrt((vectorx*vectorx)+(vectory*vectory));

return norm;
}

float CHStereo::calculateDisparityGradient(int i, int j, int p, int q)
{

float difference_in_disparity=fabs(findDeltad(i,j,p,q));
```

```
float cyclopean_seperation=cyclopeanSeperation(i,j,p,q);
if(cyclopean_seperation==0) return -1;
else return (difference_in_disparity/cyclopean_seperation);
}

void CHStereo::init(CFeatures* cf1,CFeatures* cf2,HRBitmap* cb1,HRBitmap*
cb2,CFundamental2* CFun)
  {

  m_cfun=CFun;

  init(cf1,cf2,cb1,cb2);


  }

void CHStereo::init(CFeatures* cf1,CFeatures* cf2,HRBitmap* cb1,HRBitmap* cb2){


m_leftfeature=cf1;
  m_rightfeature=cf2;
  m_leftimage=cb1;
  m_rightimage=cb2;

  CHSDialog myd;
  if(myd.DoModal()==IDOK)
    {

    m_operationMode=myd.m_mode;


    this->m_activationalpha=myd.m_alpha;
    this->m_teta=myd.m_teta;
    this->m_omega=myd.m_omega;

    ON=atoi( (LPCTSTR)myd.m_on );
    OFF=atoi( (LPCTSTR)myd.m_off );

    this->m_maxiterations=atoi( (LPCTSTR)myd.m_maxiterations);
    this->m_stableiterations=atoi( (LPCTSTR)myd.m_iterations);

    m_updatingmode=myd.m_updatingmode;
    m_showenergywindow=myd.m_energygraph;
    m_multiplematches=myd.m_showmultiplematch;
    m_showactivationfunction=myd.m_showactivation;
    m_tellenergyincrease=myd.m_energyfluctuate;
    m_writeweights=myd.m_writeweightmatrix;
    m_writeactivations=myd.m_writeactivations;
    m_objectivefunction=myd.m_objectivefunction;
    m_offset=myd.m_offset;
    m_gaussiandeviation=myd.m_gaussian;
    m_initialneuronvalue=myd.m_initialval;
```

```
    m_scale=myd.m_scale;
    m_compatibilitytype=myd.m_compatibilityfunctiontype;
    m_T=myd.m_T;

    m_compatibilityshow=myd.m_compatshow;

    m_inittype=myd.m_inittype;
    m_initvalue=myd.m_initvalue;
    m_maxdisp=myd.m_maxdisp;
    }

  }

void CHStereo::initCorrelate(CHopfield *hopfield)
{
int i,j;
float cr;
float one=1;
float two=2;
CPoint pl,pr;
for( i=1;i<=hopfield->m_height;i++){
  for( j=1;j<=hopfield->m_width;j++) {

pl=getLeftImagePoint(i,j);
pr=getRightImagePoint(i,j);

    cr=matchParallelCorrelate(m_leftimage,m_rightimage,pl,pr,5);

    cr+=one;
    cr/=two;
    hopfield->m_neuronGrid[i][j].activation=cr;
    }
  }


}
float CHStereo::generalMiu(int i, int j, int p, int q)
{

if(m_operationMode!=3)
return (-findMiu(i,p)-findMiu(j,q));//feature versus feature formulation
else
return (-findMiu(j,q));//feature versus disparity formulation
}

CPoint CHStereo::getRightImagePoint(int i,int j)
{
//this function and the next function return a feature point corresponding to
//integer J if this is a feature-vs-feature formulation, otherwise it willl return a
//point based on the disparity of the left image shifted by (j-1) points
CPoint right;
if(m_operationMode!=3){
```

```
right.x=m_rightfeature->features.x[j];
right.y=m_rightfeature->features.y[j];

  }
else
  {

right.x=m_leftfeature->features.x[j]+(i-1);
right.y=m_leftfeature->features.y[j];

  }

return right;
}

CPoint CHStereo::getLeftImagePoint(int i,int j)
{
CPoint left;
if(m_operationMode!=3){
left.x=m_leftfeature->features.x[i];
left.y=m_leftfeature->features.y[i];

  }
else
  {

left.x=m_leftfeature->features.x[j];
left.y=m_leftfeature->features.y[j];

  }

return left;

}

float CHStereo::scaleweight(int i, int j, int p, int q, float weight)
{//find distance form i to p, the distance between the points int eh left image, and scale the
wights accordingly
//this should be done since points that are too far should not provide support for each other
if(m_scale==FALSE) return weight;
//if its the same point make the weight zero
if(getLeftImagePoint(i,j).x==getLeftImagePoint(p,q).x &&
getLeftImagePoint(i,j).y==getLeftImagePoint(p,q).y) return 0;
float
distance1=findDisparity(getLeftImagePoint(i,j).x,getLeftImagePoint(i,j).y,getLeftImagePoint(p,q).x
,getLeftImagePoint(p,q).y);
float temp1=(float)0.2*(distance1-(float)40.00);
temp1=1+(exp(temp1));
temp1=((float)1.00)/temp1;
return (weight*temp1);
}
```

```
// HStereo.h: interface for the CHStereo class.
//
//////////////////////////////////////////////////////////////////////

#if
!defined(AFX_HSTEREO_H__DE2811D3_39BC_4B3D_952C_2BC27A207B1F__INCLUDE
D_)
#define
AFX_HSTEREO_H__DE2811D3_39BC_4B3D_952C_2BC27A207B1F__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "HRBitmap.h"
#include "Features.h"
#include "Hopfield.h"
#include "Fundamental2.h"
//this is the hopfield stereo class that should take two images
//and two feature sets as input and produce world coordinates as output
class CHStereo
{
public:
  float scaleweight(int i,int j,int p,int q,float weight);
  CPoint getRightImagePoint(int i,int j);
  CPoint getLeftImagePoint(int i,int j);
  float generalMiu(int i,int j,int p, int q);
  void initCorrelate(CHopfield* hopfield);
  void init(CFeatures* cf1,CFeatures* cf2,HRBitmap* cb1,HRBitmap* cb2,CFundamental2*
CFun);
  void init(CFeatures* cf1,CFeatures* cf2,HRBitmap* cb1,HRBitmap* cb2);
  float calculateDisparityGradient(int i,int j,int p,int q);
  float cyclopeanSeperation(int i, int j, int p, int q);


  float calculateWeight_3(int i,int j,int p,int q);
  float calculateWeight_2(int i,int j,int p,int q);
  float calculateWeight_1(int i,int j,int p,int q);
  float zeroWeights(CHopfield* hopfield);

  int m_objectivefunction;
  int m_operationMode;
  int m_maxiterations;
  int m_stableiterations;
  int m_inittype;
  float m_maxdisp;
  int m_compatibilitytype;
  float m_initvalue;
  float m_activationalpha;
  float m_T;
```

```
float ON;
float OFF;
CFundamental2* m_cfun;
BOOL m_showenergywindow;
BOOL m_scale;
BOOL m_multiplematches;
BOOL m_showactivationfunction;
BOOL m_tellenergyincrease;
float m_gaussiandeviation;
float m_initialneuronvalue;
BOOL m_writeweights;
BOOL m_writeactivations;
float m_offset;
float m_teta;
float m_omega;
float m_compatibilityshow;
int m_updatingmode;

void drawFunction(CHopfield *hopfield,int functiontype);
float findEnergyFromFile(char* filename);
int pruneNetwork_3(CHopfield *hopfield);
int pruneNetwork_4(CHopfield *hopfield);
void setAllWeights(CHopfield* hopfield,float value);
void sortFeatures(PointMatch* matches);
void sortFeaturesFvsF(PointMatch* matches);//F vs F = features versus features
void sortFeaturesFvsDS(PointMatch* matches);//F vs DS= features versus disparity
float findMiu(int i,int j);
float compatibility(float X);
float compatibility_1(float X);
float compatibility_2(float X);
float findDeltaD(int i,int j,int p,int q);
float findDeltad(int i,int j,int p,int q);
PointMatch* formMatches(CHopfield* hopfield);
PointMatch* formMatchesFvsF(CHopfield *hopfield);
PointMatch* formMatchesFvsDS(CHopfield *hopfield);
float findDisparity(float xl,float yl,float xr,float yr);
int pruneNetwork_1(CHopfield* hopfield);
int pruneNetwork_2(CHopfield* hopfield);
void setBiasAll(CHopfield* hopfield,float value);
float calculateWeight(int i,int j,int p,int q);
float calculateWeight(int i,int j,int p,int q,float wvalue);
void doUpdateNetwork(CHopfield* hopfield);
void doUpdateNetwork1(CHopfield* hopfield);//regular updating
void doUpdateNetwork2(CHopfield* hopfield);//winer takes all
void setWeights(CHopfield* hopfield);
void correspond(void);
enum {contineous, binary};
CFeatures* m_leftfeature;
CFeatures* m_rightfeature;
HRBitmap* m_leftimage;
HRBitmap* m_rightimage;
CHStereo();
```

```
 CHStereo(CFeatures* cf1,CFeatures* cf2,HRBitmap* cb1,HRBitmap* cb2,CFundamental2*
CFun);
 CHStereo(CFeatures* cf1,CFeatures* cf2,HRBitmap* cb1,HRBitmap* cb2);//constructor
 virtual ~CHStereo();

};

#endif //
!defined(AFX_HSTEREO_H__DE2811D3_39BC_4B3D_952C_2BC27A207B1F__INCLUDE
D_)
```

// Hopfield.cpp: implementation of the CHopfield class.
//The Hopfield network class, uses the Neuron class

```
//////////////////////////////////////////////////////////////////////

#include "stdafx.h"

#include "Hopfield.h"
#include <stdlib.h>
#include <time.h>
#include "MATX.h"

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif

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

CHopfield::CHopfield()
{
//initialize random number generator
srand( (unsigned)time( NULL ) );

}
CHopfield::CHopfield(int m,int n,float on,float off,int mode,float alpha,int inittype,float
initvalue,float inintval,float gaussian,int updatingmode,int disable)
{
srand( (unsigned)time( NULL ) );
 m_height=m ;
 m_width=n ;
flag_disable=disable;
m_updatingmode=updatingmode;
m_initialEnergyFlag=0;
m_currentEnergy=0;
ON=on;
OFF=off;
m_inittype=inittype;
```

```
m_initvalue=initvalue;
m_mode=mode;
m_alpha=alpha;
m_gaussiandev=gaussian;
m_neuroninitvalue=initvalue;
m_neuronGrid=initializeNeurons(m,n);

}



CHopfield::~CHopfield()
{
int i;
if(flag_disable!=1) {

 for( i=1;i<=m_height;i++)
   for(int j=1;j<=m_width;j++)
   m_neuronGrid[i][j].deAlloc();
 }



 for( i=0; i<m_height+2; i++)
  {
  delete [] m_neuronGrid[i];
  }


delete []  m_neuronGrid;
}



void CHopfield::clear()
{
 lastChangedNeuron=0;
 m_initialEnergyFlag=0;
 m_currentEnergy=0;



//clear all values from the neurons
 for(int i=1;i<=m_height;i++)
   for(int j=1;j<=m_width;j++)
     m_neuronGrid[i][j].clear();
}

CNeuron** CHopfield::initializeNeurons(int m, int n)
{
```

```
//create a grid of neurons and set all its values to zero
int i,j,p,q;
CNeuron** neurons;
neurons= new CNeuron* [m+2];

 for( i=0; i<m+2; i++){
   neurons[i] = new CNeuron[n+2];
  }



for( i=1;i<=m;i++){
   for( j=1;j<=n;j++){
     if(flag_disable==1)
       neurons[i][j].disable();
     else {
     neurons[i][j].setGridsize(m,n);
     neurons[i][j].setOnOFF(ON,OFF);

neurons[i][j].setMode(m_mode,m_alpha,m_inittype,m_initvalue,m_neuroninitvalue,m_gaussiande
v);
     neurons[i][j].clear();//initilize the neuron
       }
     }
  }
//now connect the neurons together

if(flag_disable!=1){
for( i=1;i<=m_height;i++)
  for( j=1;j<=m_width;j++)
    for( p=1;p<=m_height;p++)
      for( q=1;q<=m_width;q++)
        neurons[i][j].m_connection[p][q].neuron=&neurons[p][q];
  }
       return(neurons);
}

void CHopfield::writeWeightstoFile(char *filename,int linenum)
{
 if(flag_disable==1) return;


char temp1[60];
char temp2[30];
strcpy(temp1,filename);
_itoa( linenum, temp2, 10 );
strcat(temp1,temp2);
strcat(temp1,".txt");


writeWeightstoFile(temp1);
```

```
}

void CHopfield::writeWeightstoFile(char *filename)
{

 if(flag_disable==1) return;

FILE* tempf;

tempf=fopen(filename,"wt");

  for( int i=1;i<=m_height;i++)
    for( int j=1;j<=m_width;j++)
      for( int p=1;p<=m_height;p++)
        for( int q=1;q<=m_width;q++)
          fprintf(tempf,"W %d %d %d %d=
%f\n",i,j,p,q,m_neuronGrid[i][j].m_connection[p][q].weight);

fclose(tempf);

}

float CHopfield::findEnergyDiscrete()
{
//calcuulate the energy of the discrete hopfiled network
int i,j,p,q;


float energy1=0;
float energy2=0;

//firsst term
for( i=1;i<=m_height;i++)
  for( j=1;j<=m_width;j++)
    for( p=1;p<=m_height;p++)
      for( q=1;q<=m_width;q++)

  energy1+=m_neuronGrid[i][j].m_connection[p][q].weight*m_neuronGrid[i][j].activation*m_neu
ronGrid[p][q].activation;


    energy1*=((float)(((float)-1.00)/((float)2.00)));

    //now calculate the energy due to bias

for( i=1;i<=m_height;i++)
  for( j=1;j<=m_width;j++)
    energy2+=m_neuronGrid[i][j].bias*m_neuronGrid[i][j].activation;


  return (energy1-energy2);
```

```
}

float CHopfield::setConnectionValue(int i, int j, int p, int q, float value)
{
float oldvalue1,oldvalue2;

oldvalue1=m_neuronGrid[i][j].m_connection[p][q].weight;
oldvalue2=m_neuronGrid[p][q].m_connection[i][j].weight;
if(oldvalue1!=oldvalue2) AfxMessageBox("Network is not symmetric"); //just making sure
network is symmetric, although unnecessary


//now set the connection values, and make sure theyre symmetric
m_neuronGrid[i][j].m_connection[p][q].weight=value;
m_neuronGrid[p][q].m_connection[i][j].weight=value;


m_neuronGrid[i][j].m_connection[p][q].done=1;
m_neuronGrid[p][q].m_connection[i][j].done=1;
return oldvalue1;
}
void CHopfield::alignWeights()
{
 if(flag_disable==1) return;

int i,j,p,q;
//make symmetric, in case used random initilazation
for( i=1;i<=m_height;i++)
  for( j=1;j<=m_width;j++)
    for( p=1;p<=m_height;p++)
      for( q=1;q<=m_width;q++)

  m_neuronGrid[i][j].m_connection[p][q].weight=m_neuronGrid[p][q].m_connection[i][j].weight;


for( i=1;i<=m_height;i++)
  for( j=1;j<=m_width;j++)
    m_neuronGrid[i][j].m_connection[i][j].weight=0;
}
float CHopfield::setBiasValue(int i, int j, float value)
{
//use this function to set the value of a neuron's bias
float oldvalue=m_neuronGrid[i][j].bias;
m_neuronGrid[i][j].bias=value;
return oldvalue;
}

float CHopfield::updateNetwork(int n)
{
 if(m_updatingmode==0)//regular updating
  return updateNetwork1(n);
```

```
 else
  return updateNetwork2(n);//winner takes all
}
float CHopfield::updateNetwork1(int n)
{
//update the network n times using asynchronous updating scheme
  if(m_initialEnergyFlag==0) initializeEnergyValue();

  float delta_En;

for(int i=1;i<=n;i++)
  {

  int i_index=(int)randomGen(1,m_height,1);//create random indices for random updating
  int j_index=(int)randomGen(1,m_width,1);//create random indices for random updating

    //now update given neuron
  previous=m_neuronGrid[i_index][j_index].activation;
  delta_En=m_neuronGrid[i_index][j_index].update();//now update
  present=m_neuronGrid[i_index][j_index].activation;
  //if(present<0.0001) m_neuronGrid[i_index][j_index].activation=0
  //set the last changed neuron
  if(previous!=present)
  updateEnergyValue(delta_En,previous,present);
  //find new energy now
  }

return m_currentEnergy;

}
float CHopfield::updateNetwork2(int n)
{
//winner takes all

int i;
float oldvalue;
float delta_En;
float maxinput=-10000;
int rowmax=-1;
for( i=1;i<=m_height;i++)
  {
  oldvalue=m_neuronGrid[i][n].activation ;
  delta_En=m_neuronGrid[i][n].update();
  m_neuronGrid[i][n].activation =oldvalue;
  if(delta_En>maxinput)
    {
    maxinput=delta_En;
    rowmax=i;
    }
  }
```

```
for( i=1;i<=m_height;i++)
  {
 if(i!=rowmax)
   m_neuronGrid[i][n].activation=0;
 else
   m_neuronGrid[i][n].activation=1;
 }
m_currentEnergy=findEnergyDiscrete();
return m_currentEnergy;
}
void CHopfield::WriteActivations(char *filename)
{
FILE* tempf;
tempf=fopen(filename,"wt");
  fprintf(tempf,"M=[ ");
  for( int i=1;i<=m_height;i++)
    {
    if(i!=1) fprintf(tempf," ;\n");
    for( int j=1;j<=m_width;j++)
      {
      fprintf(tempf,"%f  ",m_neuronGrid[i][j].activation);
      }
    }
fprintf(tempf," ] ");
fclose(tempf);
}
void CHopfield::WriteActivations(char *filename,int linenum)
{
char temp1[60];
char temp2[30];
strcpy(temp1,filename);
_itoa( linenum, temp2, 10 );
strcat(temp1,temp2);
strcat(temp1,".txt");
WriteActivations(temp1);

}
void CHopfield::updateEnergyValue(float deltaE,float pastactivation,float presentactivation)
{
//calcuulate the energy of the discrete hopfiled network
float energy_change=0;
float activationchange;

float energy1=deltaE;
energy1*=(float)-1.0;
 activationchange=presentactivation-pastactivation;

  m_currentEnergy+=(activationchange*energy1);
}

void CHopfield::initializeEnergyValue()
{
```

```
if(m_initialEnergyFlag==0)
  {
  m_currentEnergy=findEnergyDiscrete();
  m_initialEnergyFlag=1;
  }
}
void CHopfield::disable()
{
flag_disable=1;
}
```

// Hopfield.h: interface for the CHopfield class.

```
//
//////////////////////////////////////////////////////////////////////

#if
!defined(AFX_HOPFIELD_H__A707EAAD_0FEB_49DF_85E5_83213D274059__INCLUD
ED_)
#define
AFX_HOPFIELD_H__A707EAAD_0FEB_49DF_85E5_83213D274059__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "Neuron.h"

class CHopfield
{
public:
  void disable(void);

  void initializeEnergyValue(void);
  void updateEnergyValue(float deltaE,float pastactivation,float presentactivation);
  void WriteActivations(char* filename);
  void WriteActivations(char* filename,int linenum);
  float updateNetwork(int n);
  float updateNetwork1(int n);
  float updateNetwork2(int n);//winner takes all
  float setBiasValue(int i,int j,float value);
  void alignWeights(void);
  float setConnectionValue(int i,int j,int p,int q, float value);
  float findEnergyDiscrete(void);
  void writeWeightstoFile(char* filename);
  void writeWeightstoFile(char* filename,int linenum);
  CNeuron** initializeNeurons(int m,int n);
  void clear(void);
  CHopfield();
  CHopfield(int m,int n,float on,float off,int mode,float alpha,int inittype,float initvalue,float
```

```
inintval,float gaussian,int updatingmdoe, int disable);//constructor
  virtual ~CHopfield();
  CNeuron **m_neuronGrid;
  float m_gaussiandev;
  float m_neuroninitvalue;
  int m_inittype;
  float m_initvalue;
  float ON;
  float OFF;
  int m_mode;
  float m_alpha;
  int m_updatingmode;
  int m_height,m_width;
  int m_initialEnergyFlag;
  float previous;
  float present;
  float lastChangedNeuron;
  float m_currentEnergy;
  int flag_disable;
};

#endif //
!defined(AFX_HOPFIELD_H__A707EAAD_0FEB_49DF_85E5_83213D274059__INCLUD
ED_)
```

// Neuron.cpp: implementation of the CNeuron class.
//neuron class

```
//////////////////////////////////////////////////////////////////////

#include "stdafx.h"

#include "Neuron.h"
#include "MATX.h"
#include <time.h>

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif

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

CNeuron::CNeuron()
{

flag_disable=0;
}
CNeuron::~CNeuron()
```

```
{//deallocate the grid of connections


}

//set the number of neurons in the grid mxn and initlize the weights
void CNeuron::setGridsize(int height,int width)
{
m_gridsizeHeight=height;
m_gridsizeWidth=width;
//now allocate the connections ( a 2D grid)
m_connection=new connections* [m_gridsizeHeight+2];

 for(int i=0; i<m_gridsizeHeight+2; i++)
 {
  m_connection[i] = new connections[m_gridsizeWidth+2];
 }
}
void CNeuron::clear()
{
 if(flag_disable==1) return;
//initialzie the members
if(m_initmode==0) activation=OFF;
if(m_initmode==1) activation=ON;
if(m_initmode==2) activation=randomizeActivation(2);
if(m_initmode==3) activation=randomizeActivation(3);
if(m_initmode==4) activation=m_initvalue;
if(m_initmode==5) activation=randomizeActivation(4);
if(m_initmode==6) activation=OFF;
//initialize the connection weights
for(int i=0;i<=m_gridsizeHeight;i++) {
for(int j=0;j<=m_gridsizeWidth;j++) {
m_connection[i][j].weight=0;
m_connection[i][j].done=0;

 }
 }


}

float CNeuron::update()
{
//update returns the previous value of the neuron, this is needed in the energy function of the
hopfield class
float pastvalue=activation;
float input=0;

for(int i=1;i<=m_gridsizeHeight;i++) {
 for(int j=1;j<=m_gridsizeWidth;j++) {
input+=m_connection[i][j].weight*m_connection[i][j].neuron->activation;
 }
```

```
  }
input+=bias;
activation=activationFunction(input);
return input;
}
float CNeuron::activationFunctionContineous(float a_input)
{
float temp;
temp=(tanh(a_input/m_alpha))+1;
temp=temp*((float)0.5);
return temp;
}
float CNeuron::activationFunctionDiscrete(float a_input)
{
if(a_input>0) return ON;
if(a_input<0) return OFF;
else return activation;
}
float CNeuron::activationFunction(float a_input)
{
if(m_mode==0) return activationFunctionDiscrete(a_input);
if(m_mode==1) return activationFunctionContineous(a_input);
else return activationFunctionContineous(a_input);
}
float CNeuron::randomizeActivation(int mode)
{
 if(flag_disable==1) return 0;
float random=randomGen(OFF,ON,0);
if (mode!=2 && mode!=3 && mode!=4) return mode; //static initialization (non-random)
//discrete random
if (mode==2)
 return rint(random);
//contineous random
if(mode==3) return random;
if(mode==4) return (m_initvalue2+ randomGen(0,m_gaussian,0) );
else {AfxMessageBox("incorrect function usage"); return 0;}
}

void CNeuron::deAlloc()
{
  for(int i=0; i<m_gridsizeHeight+2; i++)
    {
    delete [] m_connection[i];
    }
delete []  m_connection;
}
void CNeuron::setOnOFF(float on, float off)
{
ON=on;
OFF=off;
}
```

```
void CNeuron::setMode(int mode, float alpha,int initializationmode,float initvalue,float
initval2,float gaussian)
{
//set on and OFF first then call this function sicne this function might need to use the values for
on and off
m_mode=mode;//0 is binary and 1 is contineous
m_alpha=alpha;
m_initmode=initializationmode;//0 is off 1 is on and 2 is random binary 3 is random contineous
and 4 is by value
m_initvalue=initvalue;
m_initvalue2=initval2;
m_gaussian=gaussian;
}

void CNeuron::disable()
{
flag_disable=1;
// Neuron.h: interface for the CNeuron class.
//
//////////////////////////////////////////////////////////////////////

#if
!defined(AFX_NEURON_H__85214CF3_6BB0_43B3_8BDF_946DCE05B5BA__INCLUDE
D_)
#define
AFX_NEURON_H__85214CF3_6BB0_43B3_8BDF_946DCE05B5BA__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
#include <math.h>
#include <stdlib.h>


class CNeuron
{
public:
  void disable(void);
  void setMode(int mode,float alpha,int initializationmode,float initvalue,float initval2,float
gaussian);
  void setOnOFF(float on,float off);
  void deAlloc(void);
  float randomizeActivation(int mode);
  float activationFunctionDiscrete(float a_input);
  float activationFunctionContineous(float a_input);
  float activationFunction(float a_input);
  float update(void);
  void clear(void);
  void setGridsize(int height,int width);
  CNeuron();
  virtual ~CNeuron();
```

```
  float activation;
  float bias;
  int m_gridsizeHeight;
  int m_gridsizeWidth;
  int flag_disable;
  float ON;
  float OFF;
  float m_alpha;
  int m_initmode;
  float m_initvalue;
  float m_initvalue2;
  float m_gaussian;
  int m_mode;//0 is binary and 1 is conitineous
  int valid;
struct connections{
  float weight;
  int done;
  CNeuron* neuron;
  } ;


  connections** m_connection;

};

#endif                                                                    //
        !defined(AFX_NEURON_H__85214CF3_6BB0_43B3_8BDF_946DCE05B5
        BA__INCLUDED_)
```

VITA AUCTORIS

NAME:	Houman Rastgar
PLACE OF BIRTH:	Shiraz, Iran
YEAR OF BIRTH:	1980
EDUCATION:	McMaster University, Hamilton, Ontario 1999-2003 Bachelor of Applied Science
	University of Windsor, Windsor, Ontario 2003-2006 Master's of Applied Science