CONSTRUCTION OF A MACHINE VISION SYSTEM FOR AUTONOMOUS APPLICATIONS

by

Anthony Nicholas Lorusso

Submitted to the
Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements
for the Degree of
Master of Electrical Engineering
at the
Massachusetts Institute of Technology
May, 1996

Signature of Author_____
Depa ...puter Science
May 23, 1996

Certified by_____
Berthold Klaus Paul Horn
Thesis Supervisor

Certified b_____
David S. Kang
Project Supervisor

Accepted by_____
F. R. Morganthaler
Chair, Department Committee on Graduate Students

CONSTRUCTION OF A MACHINE VISION SYSTEM FOR AUTONOMOUS APPLICATIONS

by
Anthony Nicholas Lorusso

Submitted to the Department of Electrical Engineering and Computer Science on May 23, 1996 in partial fulfillment of the requirements for the Degree of Master of Electrical Engineering.

Abstract

The Vision System is a completed prototype that generates navigation and obstacle avoidance information for the autonomous vehicle it is mounted upon. The navigational information consists of translation in three axes and rotation around these same axes constituting six degrees of freedom. The obstacle information is contained in a depth map and is produced as a by-product of the navigation generation algorithm. The coarse prototype generates data that is accurate to within 15%. The design methodology and construction of the prototype is carefully outlined. All of the schematics and code are included within the thesis.

Assignment of Copyright to Draper Laboratory

This thesis was supported by The Charles Stark Draper Laboratory, Inc. through the Intelligent Unmanned Vehicle Center (IUVC). Publication of this thesis does not constitute approval by The Charles Stark Draper Laboratory, Inc. of the findings or conclusions contained herein. It is published for the exchange and stimulation of ideas.

I hereby assign my copyright of this thesis to The Charles Stark Draper Laboratory, Inc., Cambridge, Massachusetts.

Anthony Nicholas Lorusso

Permission is hereby granted by The Charles Stark Draper Laboratory, Inc. to the Massachusetts Institute of Technology to reproduce part and/or all of this thesis.

Acknowledgments

I would like to thank

Matthew Fredette, James Guilfoy, Berthold Horn, Anne Hunter, William Kaliardos, David Kang, Dolores Lorusso, Grace Lorusso, Donald Lorusso Sr., Stephen Lynn, Eric Malafeew, William Peake, Ely Wilson.

Dedication

This thesis is dedicated to the people whom have made all of the wonderful things in my life possible.

To my wife,

Dolores Marie Lorusso

my father in law,

Anthony George Sauca

and

Frank & Concetta Ternullo.

Table of Contents

INTRODUCTION

The Vision System described within this thesis has been designed specifically for autonomous applications where an autonomous application is any system that can perform a function without assistance. A few good examples of autonomous systems are security robots, house alarms, or blind persons with seeing eye dogs -these particular examples would all benefit from the system described here.

The system is tailored for stereo or monocular camera arrangements, low power consumption, small packaging, and faster processing than any similar commercial Vision System available today. The ultimate goal of this thesis is to describe the construction of a complete vision that can be mounted on an autonomous vehicle. The prototype described here will serve as a "proof of concept" system for the new ideas introduced within it; the system can be considered as an iteration in a much larger process to produce a self contained vision processing unit, perhaps even a single application specific integrated circuit.

This thesis is the third in a series of documents produced by the Intelligent Unmanned Vehicle Center (IUVC) of The Charles Stark Draper Laboratory in Cambridge Massachusetts that focus on the topic of machine vision. The first document is "Using Stereoscopic Images to Reconstruct a Three -Dimensional Map of Martian or Lunar Terrain" by Brenda Krafft and Homer Pien. The second document is "A Motion Vision System for a Martian Micro-Rover" by Stephen William Lynn. The IUVC project was established in 1991 with the participation of undergraduate and graduate students from the Massachusetts Institute of Technology and other universities local to Boston.

IUVC has given birth to many autonomous robots and vehicles since its establishment. The first series of robots, the MITy's, were designed for Lunar and Mars exploration. IUVC's most notable microrover, MITy2, has a cameo appearance in the IMAX film "Destiny in Space." Currently IUVC is developing more sophisticated ground, aerial, and water crafts.

"Construction of a Machine Vision System for Autonomous Applications," builds very closely on IUVC's second vision system document, "A Motion Vision System for a Martian Micro-Rover."

The purpose of this paper is to clearly explain the prototype Vision System so that others, if desired, may be able to recreate it. Since, however, the paper will dwell heavily upon the work that was actually implemented in the Vision System only a few alternate solutions will be discussed.

As you will see, the Vision System described here is simple, novel, flexible, expandable, and fast.

CHAPTER ONE
THE THEORY BEHIND THE SYSTEM

This chapter will provide all of the background theory necessary to understand how the vision system works. The natural progression of the chapter is to build the mathematical models and equations in the context that they will be used.

The chapter starts by defining the coordinate system that will be used, followed by a derivation of the Longuet Higgins & Pradzny Equations. The rest of the chapter describes the pattern matching and motion calculation algorithms in conjunction with their supporting formulas. The supporting formulas include the Least Squares derivation for calculating motion parameters , the optimized depth formula for updating the depth map.

The Coordinate System

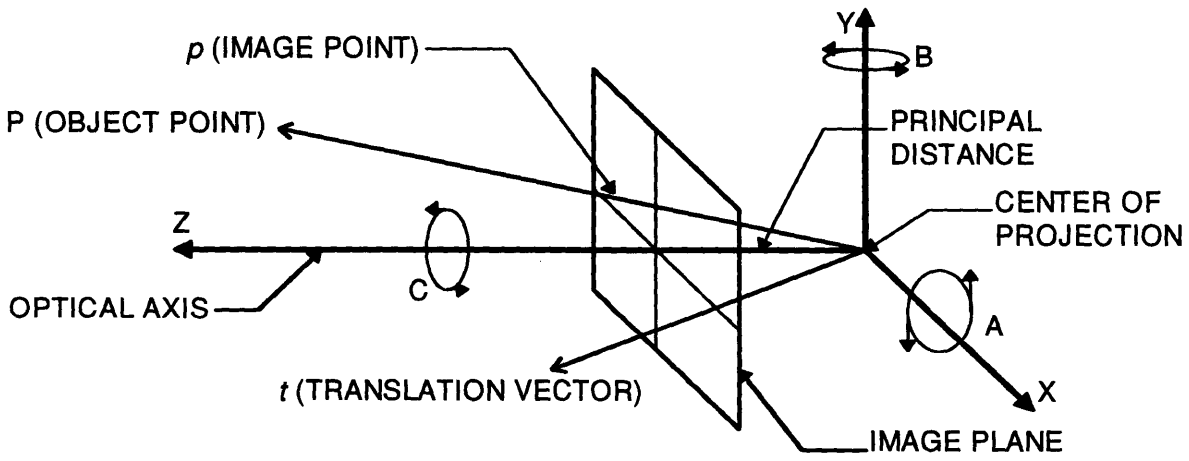


Figure 1. 1 The Coordinate System

The coordinate system defined here is not nearly as complicated as it appears. The origin is renamed the Center of Projection, since any ray of light coming from an object passes through it. All points that are on objects seen by our system are generally labeled P. Conversely all object points correspond to a unique image point on the Image Plane. An

image point is where the ray connecting an object point to the Center of Projection intersects the Image Plane; these are denoted by a lower case script p. The Z-axis is called the Optical Axis, and the distance along the Optical Axis between the Image Plane and the Center of Projection is the Principal Distance. Any motion of our coordinate system, along the axes is described by a translation vector t. The right handed angles of rotation along each of the axes are A, for the X-axis, B, for the Y-axis, and C for the Optical Axis.

Derivation of the Longuet Higgins & Pradzny Equations

An easy way to picture the coordinates is as though the Charge Coupled Device (CCD) Camera is permanently mounted at the Center of Projection, and that the Image Plane is the camera's internal CCD sensor. (See Figure 1.2).

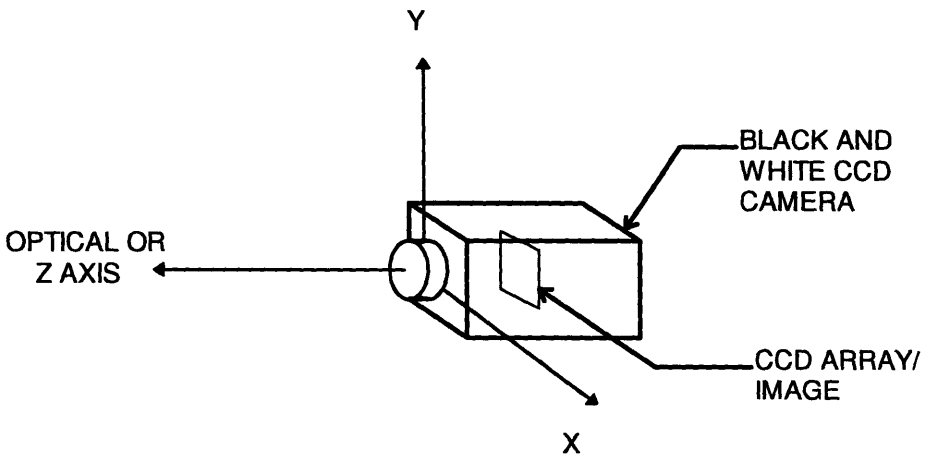


Figure 1.2 CCD Camera Placement

The Optical Axis goes through the center of the camera and points out of the camera lens; this is the exact way that the coordinate system and camera will be used. Note that throughout this chapter, lowercase x & y refer to coordinates on the Image Plane.

A camera cycle is simple; an image from the camera is stored, the camera is moved, and a second image from the camera is also stored. The camera motion between images can be described in translation and rotation vectors. It is a nice twist to note that if the environment moved instead of the camera, it can still be described in the same way

with just a change of sign. The structures we will use to describe motions in our coordinate system are,

The translation vector,

$$t = \begin{bmatrix} U & V & W \end{bmatrix}^T$$

Equation 1.1 Translation Vector

where U, is translation in the X-axis, V is translation in the Y-axis, and W is translation in the Optical, or Z-axis. The rotation vector,

$$r = \begin{bmatrix} A & B & C \end{bmatrix}^T$$

Equation 1.2 Rotation Vector

where A, is the rotation in radians around the X-axis, B is the rotation around the Y-axis, and C is the rotation around the Optical, or Z-axis. It is importnat to note that in practice we will expect these values to be small angular increments.

In Equation 1.2 each of the rotations may be represented by rotation matrices [1]. The matrices for the rotations are in bold or have a subscript *m*.

For angle **A**, rotation in the Y-Z plane,

$$A_m = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos A & \sin A \\ 0 & -\sin A & \cos A \end{bmatrix}$$

Equation 1.3 Rotation Matrix for X-axis

For angle **B**, rotation in the X-Z plane,

$$B_m = \begin{bmatrix} \cos B & 0 & -\sin B \\ 0 & 1 & 0 \\ \sin B & 0 & \cos B \end{bmatrix}$$

Equation 1.4 Rotation Matrix for Y-axis

For angle C, rotation in the X-Y plane,

$$C_m = \begin{bmatrix} \cos C & \sin C & 0 \\ -\sin C & \cos C & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Equation 1.5 Rotation Matrix for Z-axis

An object point in the scene being imaged is described by coordinates in X, Y, Z,

$$P = \begin{bmatrix} X & Y & Z \end{bmatrix}^T$$

Equation 1.6 Object Point Vector

Camera motion causes an object point to appear in a new location. The new location is given by,

$$\begin{bmatrix} X' \\ Y' \\ Z' \end{bmatrix} = C_m B_m A_m \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} - t \Leftrightarrow \begin{bmatrix} X' \\ Y' \\ Z' \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} - \begin{bmatrix} U \\ V \\ W \end{bmatrix}$$

Equation 1.7 New Object Coordinate Equation

where R is the multiplication of equations 1.5, 1.4, and 1.3 respectively. After multiplication the R matrix is,

$$R=\begin{bmatrix} \cos B\cos C & \sin A\sin B\cos C+\cos A\sin C & -\cos A\sin B\cos C+\sin A\sin C \\ -\cos B\sin C & -\sin A\sin B\sin C+\cos A\cos C & \cos A\sin B\cos C+\sin A\cos C \\ \sin B & -\sin A\cos B & \cos A\cos B \end{bmatrix}$$

Equation 1.8 The Complete Rotation Matrix

The order of multiplication of the rotation matrices is very important. In an attempt to backward calculate for C, B, and A the order of multiplication would have to be known. The rotation matrices have another concern in addition to their order of multiplication, the matrices have non-linear elements.

The ability to make the elements of the rotation matrices linear will allow them to be solved for easier. The best way to do this is to use a Taylor expansion to simplify our equations. Recall the Taylor expansion is,

$$f(x)=f(a)+(x-a)f'(a)+\frac{(x-a)^2}{2!}f''(a)+\frac{(x-a)^3}{3!}f'''(a)+\ldots+\frac{(x-a)^n}{n!}f^{(n)}(a)+\ldots$$

Equation 1.9 Taylor Expansion [2]

We will choose our expansion point to be P=(A=0, B=0, C=0), and our second-order terms are negligible. Thus, the Taylor expansion is,

$$R=I+\left.\frac{\partial R}{\partial A}\right|_P A_m+\left.\frac{\partial R}{\partial B}\right|_P B_m+\left.\frac{\partial R}{\partial C}\right|_P C_m$$

Equation 1.10 Taylor Expansion of R Around P

where the partial derivatives are,

$$\frac{\partial R}{\partial A}=\begin{bmatrix}0 & \cos A\sin B\cos C-\sin A\sin C & \sin A\sin B\cos C+\cos A\sin C\\ 0 & -\cos A\sin B\sin C-\sin A\cos C & -\sin A\sin B\sin C+\cos A\cos C\\ 0 & -\cos A\cos B & -\sin A\cos B\end{bmatrix}$$

$$\frac{\partial R}{\partial B}=\begin{bmatrix}-\sin B\cos C & \sin A\cos B\cos C & -\cos A\cos B\cos C\\ \sin B\sin C & -\sin A\cos B\sin C & -\cos A\cos B\sin C\\ \cos B & \sin A\sin B & -\cos A\sin B\end{bmatrix}$$

$$\frac{\partial R}{\partial C}=\begin{bmatrix}-\cos B\sin C & -\sin A\sin B\sin C+\cos A\cos C & \cos A\sin B\sin C+\sin A\cos C\\ -\cos B\cos C & -\sin A\sin B\cos C-\cos A\sin C & \cos A\sin B\cos C-\sin A\sin C\\ 0 & 0 & 0\end{bmatrix}$$

and the partial derivatives evaluated at P are conveniently,

$$\left.\frac{\partial R}{\partial A}\right|_P=\begin{bmatrix}0 & 0 & 0\\ 0 & 0 & 1\\ -1 & 0 & 0\end{bmatrix},\quad \left.\frac{\partial R}{\partial B}\right|_P=\begin{bmatrix}0 & 0 & -1\\ 0 & 0 & 0\\ 1 & 0 & 0\end{bmatrix},\quad \left.\frac{\partial R}{\partial C}\right|_P=\begin{bmatrix}0 & 1 & 0\\ -1 & 0 & 0\\ 0 & 0 & 0\end{bmatrix}$$

and so our Taylor expansion to the first order is,

$$R=I+\begin{bmatrix}0 & 0 & 0\\ 0 & 0 & A\\ 0 & -A & 0\end{bmatrix}+\begin{bmatrix}0 & 0 & -B\\ 0 & 0 & 0\\ B & 0 & 0\end{bmatrix}+\begin{bmatrix}0 & C & 0\\ -C & 0 & 0\\ 0 & 0 & 0\end{bmatrix}$$

Equation 1.11 Taylor Expansion to the First Order of R

Keep in mind that our coordinate system is perspective projection. To obtain Image Plane coordinates, *x* & *y*, normalized by the camera lens focal length, *f*, we take our world coordinates X & Y, and divide by the principal distance, which is Z:

$$\frac{x}{f}=\frac{X}{Z},\quad \frac{y}{f}=\frac{Y}{Z}$$

Finally, we can express our Image Plane motion field in terms of the indicated parameters. The motion field is made up of the motion of points in the Image Plane as the camera is translated and rotated. The components of the motion vector for a point in the Image plane are [3],

$$\frac{u}{f} = \dot{x} = \frac{\dot{X}}{Z} - \frac{\dot{Z}}{Z^2}X, \quad \frac{v}{f} = \dot{y} = \frac{\dot{Y}}{Z} - \frac{\dot{Z}}{Z^2}Y$$

Equations 1.12 Motion Field Equations

where the dotted quantities are functions of images and u is motion along the image plane x-axis, and v is motion along the image plane y-axis. Equations 1.12 were derived by the product rule of the normalized coordinate equations.

Now if we step back and examine the X, Y, Z world coordinates it is obvious that we can represent their dotted quantities or changes with respect to images as a difference,

$$\begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{Z} \end{bmatrix} = \begin{bmatrix} X' \\ Y' \\ Z' \end{bmatrix} - \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = (R - I)\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} - \begin{bmatrix} U \\ V \\ W \end{bmatrix}$$

Equation 1.13 Dotted World Coordinates

Now, plugging the Taylor approximation into Equations 1.12 and simplifying, we get,

$$\frac{u}{f} = \frac{CY - BZ - U}{Z} - \frac{X(BX - AY - W)}{Z^2}$$
$$\frac{v}{f} = \frac{AZ - CX - V}{Z} - \frac{Y(BX - AY - W)}{Z^2}$$

which simplify to the Longuet Higgins & Pradzny Equations,

$$\frac{u}{f} = \frac{\frac{xW}{f} - U}{Z} + \frac{xyA}{f^2} - \left(\left(\frac{x}{f}\right)^2 + 1\right)B + \frac{yC}{f}$$

$$\frac{v}{f} = \frac{\frac{yW}{f} - V}{Z} + \left(\left(\frac{y}{f}\right)^2 + 1\right)A - \frac{xyB}{f^2} - \frac{xC}{f}$$

Equations 1. 14 Longuet Higgins & Pradzny Equations [4]

QED

The Longuet Higgins & Pradzny equations provide a practical and clear relation between the components of the motion field in the Image Plane of a camera and the actual motion of the surrounding environment.

The Pattern Matching Algorithm

The camera cycle compares two consecutive CCD images using the pattern matching algorithm. The pattern matching algorithm produces the relative motion field that can be used to determine camera motion or gather information about obstacles in the environment, depending upon the need of the end user.

The information collected from the black and white CCD camera is stored in an n*x*m matrix of 8 bit values, where n is the horizontal resolution of the Image Plane and m is the vertical resolution of the Image Plane. The resolution of the Image Plane depends upon the number of pixels in the CCD of the camera used to image the environment. Each complete n*x*m matrix of the Image Plane is called a frame. Each 8-bit value, called a pixel, is the brightness of a particular point in the Image Plane. The range of brightness values varies between 0 and 255, with 0 being a black pixel and 255 being a white pixel. Together, all the pixels make up an image of the surrounding environment. (See Figure 1.3).

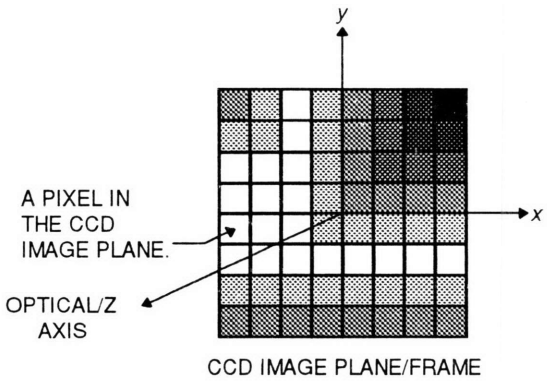


Figure 1.3 The CCD Image Plane and Frame

The operation of the pattern matching algorithm is functionally straight forward. The first frame is separated into data blocks nxn pixels in size. The second frame is separated into search windows 2nx2n pixels in size. The frame to block parsing is done such that there are the same number of data blocks and search windows. All of the data blocks have their centers at the same positions as the centers of the search windows in their respective frames. (See Figure 1.4).

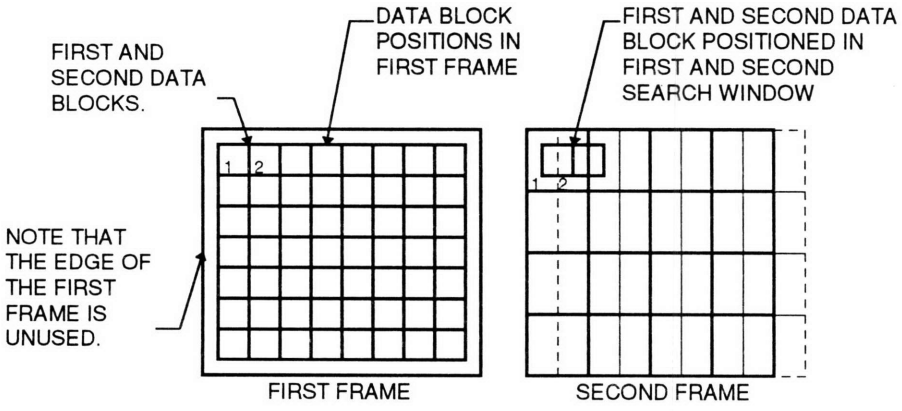


Figure 1.4 First and Second Frames

The algorithm takes the first data block and finds its best match in the first search window. The best match position is then an offset from the data blocks original position. The offset is considered to be a measurement of the local motion field, where each of its components, u & v are the components of motion in Equations 1.14, the Longuet Higgins & Pradzny equations. Each offset will be called a flow vector.

The best match is determined by the smallest error. A data block is initially placed in the upper left corner of its respective search window. The absolute value of the difference in the pixel values between the data block and search window are then summed to give the position error. The datablock is then moved to the right by one pixel and a new position error is computed. This process continues until the data block has been positioned at every possible location in its search window, See Figure 1.5. The data block location of the smallest position error is the desired value. If two position errors have the smallest value the second error computed is taken to be the best match. A description of the algorithm is,

$$for(-\frac{n}{2} \le i < \frac{n}{2}, -\frac{n}{2} \le j < \frac{n}{2}):$$

$$PE_{ij} = \sum_{y=0}^{n-1}\sum_{x=0}^{n-1} \left| SW_{x+i,y+j} - DB_{x,y} \right|$$

$$if\,(PE_{i,j} \le ME_{I,J})$$

$$then,\ ME = PE_{I=i,J=j}.$$

Equation 1. 15 Pattern Matching Algorithm [5]

where PE is the position error, SW is the search window, DB is the data block, and ME is the minimum position error. The subscripts refer to specific locations in the respective data arrays. The value n is the width and height of a data block and, for hardware reasons discussed later, the upper left pixel in the search window is given the coordinate (-n/2, -n/2). I & J are the position of the best match.

It is important to note that this algorithm does not provide us with any information about the quality of the match. Further, the algorithm only provides integer pixel resolution.

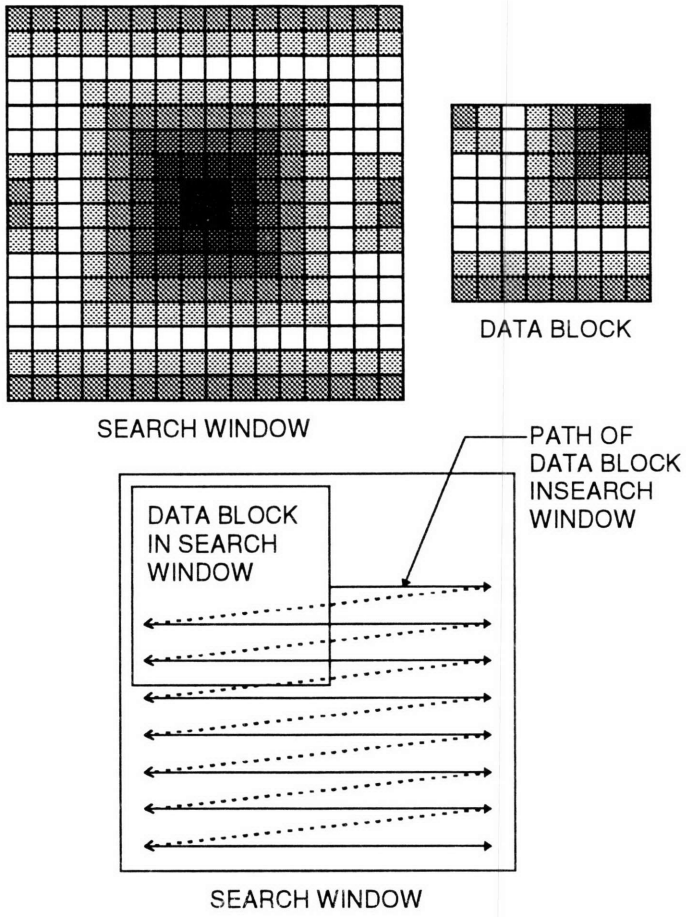


Figure 1.5 Movement of Data Block in Search Window

The position of the best match and its associated flow vector are calculated for each data block and search window pair. After all of the flow vectors have been calculated we have determined our complete motion field. It is important to note that the size of our search window places an upper bound on the magnitude of each flow vector. Hence, the amount of motion in a camera cycle is limited. It is also important to mention that we are assuming that the patterns in the frame parsed into data blocks are approximately the same patterns in the frame parsed into search windows; this is not a trivial assumption.

Motion Parameter Calculation Algorithm

The motion parameter calculation algorithm produces the final results of the vision system; It computes the 6 motion parameters described in Equations 1.1 and 1.2, and a depth map of the surrounding environment. The six motion parameters describe the net

movement of the camera in a camera cycle. The depth map contains the estimated distances from the camera to large objects in the first frame of the camera cycle; a depth estimate is computed for each data block. Therefore, the resolution of the depth map is dependent on the size of the data block; the depth map can be thought of as the low pass filtered environment. The depth map suppresses all of the sharp detail and contrast an image can provide.

The motion algorithm simultaneously solves for the motion parameters and depth map; in a sense, they are by products of one another. The algorithm is an iterative process that assumes the correct motion parameters have been achieved when the depth map values converge. The motion algorithm is,

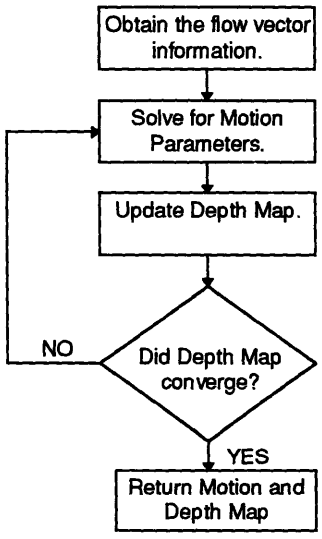


Figure 1.6 Motion Calculation Flow Chart

The first step in Figure 1.6 is to obtain the flow vector information from the pattern matching algorithm. A flow vector is formally defined as,

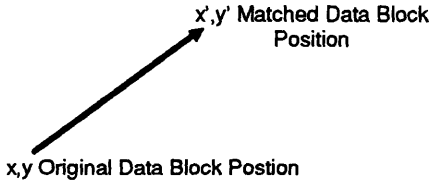


Figure 1.7 Flow Vector

and we can relate each flow vector to Equations 1.14, the Longuet Higgins & Pradzny equations by,

$$u = \dot{x} = x' - x, \quad v = \dot{y} = y' - y$$

Equations 1.16 Flow Vector Components

where the focal length has been set to one for convenience. The Longuet Higgins and Pradzny equations relate the motion field in the Image Plane to the actual motion of a camera in our coordinate system, Figure 1.1. Since the pattern matching algorithm provides us with a means of determining the motion field in the Image Plane, it is sensible that we can solve for the six parameters of camera motion, U, V, W, A, B, C. We will define a vector *m*, that contains all of the motion parameters,

$$m = \begin{bmatrix} U & V & W & A & B & C \end{bmatrix}^T$$

Equation 1.17 Motion Parameter Vector

The motion parameter vector allows us to rewrite Equations 1.14, the Longuet Higgins & Pradzny equations, for each flow vector as,

$$\begin{bmatrix} \frac{-1}{Z} & 0 & \frac{x}{Z} & xy & -(x^2+1) & y \\ 0 & \frac{-1}{Z} & \frac{y}{Z} & (y^2+1) & -xy & -x \end{bmatrix} m = \begin{bmatrix} u \\ v \end{bmatrix}$$

Equation 1.18 Matrix Form of Longuet Higgins and Pradzny

The matrix for all the flow vectors is called*A*. *A* is a 2nxm matrix of the form,

$$\begin{bmatrix} \frac{-1}{Z_1} & 0 & \frac{x_1}{Z_1} & x_1 y_1 & -\left(x_1^2+1\right) & y_1 \\ 0 & \frac{-1}{Z_1} & \frac{y_1}{Z_1} & \left(y_1^2+1\right) & -x_1 y_1 & -x_1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{-1}{Z_n} & 0 & \frac{x_n}{Z_n} & x_n y_n & -\left(x_x^2+1\right) & y_n \\ 0 & \frac{-1}{Z_n} & \frac{y_n}{Z_n} & \left(y_n^2+1\right) & -x_n y_n & -x_n \end{bmatrix} m = Am$$

Equation 1.19 Matrix of the Flow Vectors

Equation 1.19 is equal to the motion field derived from the pattern matching. Thus, an easy way of writing all the flow vectors in terms of the Longuet Higgins and Pradzny equations and setting them equal to the derived motion field is,

$$Am = \begin{bmatrix} u_1 & v_1 & u_2 & v_2 & \cdots & u_n & v_n \end{bmatrix}^T = b,$$
$$Am = b$$

Equation 1.20 Flow Vectors Set Equal to the Motion Field

where b contains the components of the motion field derived from the pattern matching algorithm.

We still, however, need to find the best way to solve for m. If estimated values were chosen for m and A, an error could be defined between the b matrix containing the components of the motion field derived from pattern matching and the b matrix calculated from the estimated values of m & A. Since we have two parameters that the error could be optimized along, it is best to minimize the sum squared error. The sum squared error for a single flow vector is,

$$SSE = \left(\hat{u} - u\right)^2 + \left(\hat{v} - v\right)^2$$

Equation 1.21 Sum Squared Error for a Single Flow Vector

where u & v caret are derived from the Longuet Higgins and Pradzny equations. The error could be optimized to either the x-component, u, or the y-component, v. The total error for all the flow vector components in the b matrix would be

$$TSSE = \sum_n \left(\hat{u}_n - u_n\right)^2 + \left(\hat{v}_n - v_n\right)^2$$

Equation 1.22 Total Sum Squared Error for All Flow Vectors

which can be also written, in matrix form, as,

$$\|Am - b\|^2 = 0$$
$$where, \|x\|^2 = x_1^2 + x_2^2 + \ldots + x_n^2$$

Equations 1.23 Matrix Form of Total Sum Squared Error

The equation $Am=b$ is overdetermined as long as we use more than six flow vectors. The beauty of the matrix form is that it can be shown to have the Least Squares solution for m.

$$m = \left(A^T A\right)^{-1} A^T b$$

Equation 1.24 Least Squares Motion Estimation Equation [6]

Thus, we can solve for our motion given an estimated depth map.

Scale Factors

It is important to understand what the computed motion parameters and depth map mean. The rotations will always be in units of radians and the translations and depths will always be in the units of the principal distance, which are meters. The translations and depth values, however, will require scale factors to make them accurate. The easiest explanation for scaling is to observe that if the external world were twice as large and our velocity through the world were twice as fast all of our equations would generate the same flow field and depth map [7]. Unfortunately, determining the scale factor requires some knowledge of the environment external to the Vision System. Also, the scale factor can be

applied in two separate ways, either to the relative depth map or to the translation values themselves. Applying the scale factor to the translations requires measuring one of the parameters on axis and dividing by the estimated value to get the scale factor and then multiplying the other translations by it. Applying the scale factor to the depth map requires measuring the depth to a particular point in a scene and dividing it by the estimated depth value at that point to get the scale factor; this scale factor would then multiply all of the other depth values. Once we have either a corrected depth map or motion parameter vector we can calculate to get the correct motion parameter vector or depth map respectively.

Updating the Depth Map

Initially, we assumed the values of our depth map were constant; this allowed the first calculation of the motion parameters using Equation 1.24. A constant depth map, however, is not representative of a typical environment. To obtain more precise motion information a more accurate depth map may be needed. Hence, local correction of the depth map using the latest motion parameters could be done by isolating our unscaled depth Z, in terms of the six motion parameters and position. The equation for Z can be derived by optimizing the sum squared error between the motion field calculated by Equations 1.14, the Longuet Higgins and Pradzny equations, and the motion field computed from the pattern Matching. It is reassuring to note that this is the same sum squared error equation, Equation 1.21, that was used to derive the equation for the global motion parameters, Equation 1.24. The sum squared error with the full Longuet Higgins and Pradzny equations inserted and the focal length set equal to one for simplicity are,

$$SSE = \left(\frac{(-U + xW)}{Z} + xyA - (x^2 + 1)B + yC - u \right)^2 + \left(\frac{(-V + yW)}{Z} + (y^2 + 1)A - xyB - xC - v \right)^2$$

to obtain a local minima we take the partial derivative with respect to Z, and set it equal to zero,

$$\frac{\partial SSE}{\partial Z} = -\frac{(-U + xW)^2}{Z} + \left(xyA - \left(x^2 + 1\right)B + yC - u\right)\left(-(-U + xW)\right) + -\frac{(-V + yW)^2}{Z} + \left(\left(y^2 + 1\right)A - xyB - xC - v\right)\left(-(V + yW)\right) = 0$$

which, after some careful manipulation becomes our local depth Z,

$$Z = \frac{(-U + xW)^2 + (-V + yW)^2}{\left(u - \left(xyA - \left(x^2 + 1\right)B + yC\right)\right)(-U + xW) + \left(v - \left(\left(y^2 + 1\right)A - xyB - xC\right)\right)(-V + yW)}$$

Equation 1.25 Update Depth Map Equation [8]

Now, we use Equation 1.25, to update the depth map, where each flow vector has a single depth value. Heel has shown sufficient convergence of the depth map within ten iterations [9]. We will use convergence of the depth map as a measure of the reliability of the motion parameters calculated using it. It is only necessary to complete a full convergence cycle for the first computation of the motion estimates. The converged depth map can be used as the a priori depth map for the next motion estimate instead of a constant value. In most instances, this will decrease the number of iterations it takes the depth map to converge.

Chapter Summary

This chapter derived all of the pertinent Machine Vision equations necessary to understand this Vision System.

The chapter opened with a description of the coordinate system and data structures used within the chapter. The coordinate system is comprised of two components, the three dimensional world coordinate axes, and the two dimensional Image Plane. The world coordinate axes are labeled X, Y, Z, and the rotations are respectively, A, B, C. The Image Plane coordinates are x, y. An object point, P, is in world coordinates

and described by Equation 1.6. An Image Point, p, is the contact location on the Image Plane of a ray that connects the Center of Projection, previously known as the origin of the world coordinate axes, and an object point. The Image Plane is actually the CCD of the camera used to view the environment. Any translation and rotation in the world coordinates is described as three dimensional vectors t & r, which are Equations 1.1 and Equations 1.2 respectively.

Immediately following the definitions of the coordinate system came the derivation of the Longuet Higgins and Pradzny equations, Equations 1.14. The Longuet Higgins and Pradzny equations provide a way of directly relating, within a scale factor, the motion field in the Image Plane to the six parameters of motion in world coordinates, where the motion field is a set flow vectors detailing how objects in the Image Plane move. A flow vector has two components, u & v which are change in x & y respectively.

Pattern Matching is used to measure the motion field between two consecutive frames. The frames are broken into manageable pieces that the pattern matching is performed on. Each small piece of the first frame is found in a corresponding larger piece of the second frame by finding the location of minimum error, called a best match. The size of the pieces that each frame is broken into determine the maximum value that a flow vector can be. The number of pieces that the frames are broken into determines the resolution of the Vision System. The Pattern Matching Algorithm is defined in Equation 1.15.

Following the pattern matching is the derivation of the equation that calculates the six motion parameters, Equation 1.24. The equation takes two sets of numbers and computes the Least Squares values of the six motion parameters. The first set of numbers is a matrix A, calculated from the depth, Z, and the x, y positions of each flow vector. The second set of numbers is a matrix b, which contains all of the flow vector components determined from pattern matching. The six motion parameters U, V, W, A, B, C, are each elements of matrix m.

The cycle to calculate the six motion parameters is an iterative algorithm that assumes that the motion parameters are correct, to within a scale factor, when the depth map converges. The cycle takes a depth map, assumed constant only on the first cycle, and is delineated in Figure 1.6.

The chapter concludes with the derivation of the equation used to update the depth map, Equation 1.25. The equation performs local depth calculation using the six global motion parameters in matrix m, and the original positions of each flow vector.

The chapter not only describes all of the major theory implemented within this Vision System, but provides a brief introduction into the field of Machine Vision; the field of vision has many more facets than those presented here.

CHAPTER TWO
PROTOTYPE VISION SYSTEM ARCHITECTURE

Chapter two presents the system architecture used in the prototype vision system developed for this thesis. The prototype serves as a proof of concept.

The chapter starts by outlining the natural way to perform image processing as described within this thesis. Each functional block of the natural image processing flow is considered and the important design issues are defined. The chapter concludes by presenting the architecture used in the prototype proof of concept system developed for this thesis.

The Natural Image Processing Flow

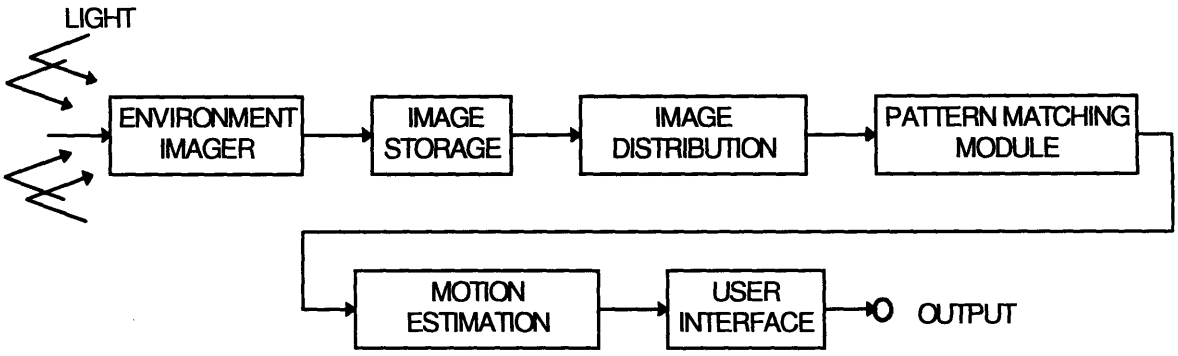


Figure 2. 1 Natural Image Processing Block Diagram [10]

The implementation of the theory developed in this thesis has a natural progression which can easily be represented as a block diagram, see Figure 2.1. The functional blocks are the Environment Imager, Image Storage, Image Distribution, Pattern Matching, Motion Estimation, and a User Interface. The following sections describe each of the functional components and their primary design concerns.

The Environment Imager

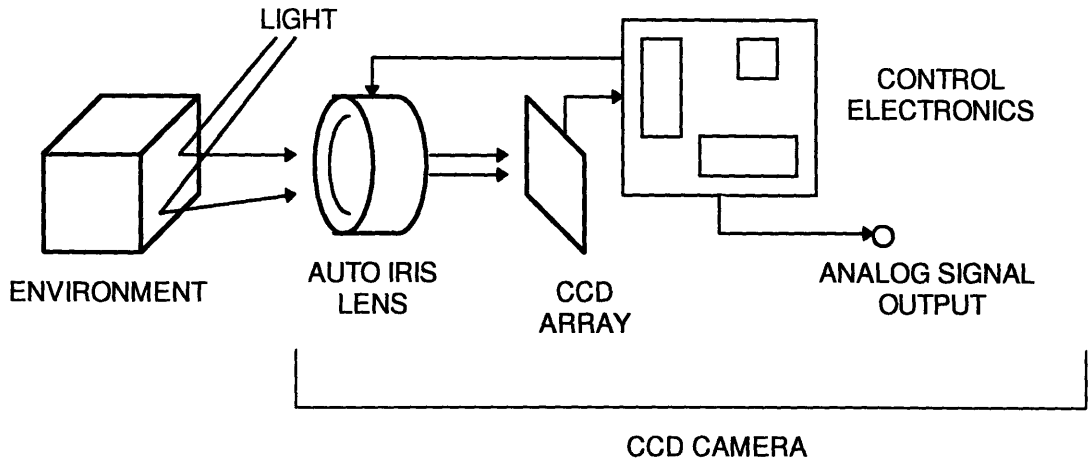


Figure 2.2 The Environment Imager

The environment imaging module is used to convert reflected light from the surroundings into usable information, see Figure 2.2. There are many techniques used to image an environment, however, the most common manner is to use a charge coupled device (CCD) camera.

A CCD camera uses a grid of photoelectric sensors that separate charge, and thus create a voltage across them when exposed to certain frequencies of light. CCD arrays are typically made from silicon and thus have the typical spectral responsivity for a silicon detector. Each photoelectric element in a CCD array is called a pixel. The camera uses a lens that focuses the environment being imaged onto the CCD. A CCD controller then shifts the voltages from the CCD elements out. In order to make a CCD camera applicable to the current video standards, the digital data shifted out of the CCD is converted into one of the standard analog formats. Some of the analog formats are NTSC, CCIR/PAL. The number of pixels in the CCD array gives the resolution of the camera; it is usually best to get a camera with a large number of pixels (i.e. high resolution.)

CCD arrays come in many different styles depending upon the application. One type of CCD array is for color cameras. A color camera has three separate analog signals; one for each of the colors red, green, blue. Although color CCD cameras can also be used

with this system, a black and white camera makes more practical sense since we are not trying to identify any objects or edges using colors, but by contrast. In a pattern matching system, a color camera requires three times as much processing as that of a black and white camera. We want to identify where different objects are and how they move. In a black and white camera colors are seen as shades of gray; this is advantageous when we make the constant pattern assumption since all of the contrast information is contained in each captured image.

The CCD array in a camera can also have pixels that are different shapes. Arrays can be purchased with hexagonal, circular, and square pixels. An array with circular pixels does not use all of the available light to generate its image data. CCD's with hexagonal pixels have a complicated geometry that must be considered when the data is processed. Therefore, the array with square pixels is the best possible choice for our application.

The CCD, however, is not our only concern about using a CCD camera as the environment imager. The amount of light that impinges on the CCD and contrast control are of equal importance.

The camera should have the ability to control the total amount of light that is focused on the CCD by opening and closing an iris; this feature is called auto iris. The auto iris controls the amount of light that enters a camera so that the full dynamic range of the CCD array can be used; it keeps the CCD from being under illuminated, or from being saturated by over illumination.

The last important feature of the environment imager is that the overall image contrast, gamma, should be able to be controlled. Many CCD cameras that are available allow the user to control gamma; this is especially important because Machine Vision may require higher contrast than the human eye finds comfortable.

The Image Storage Module

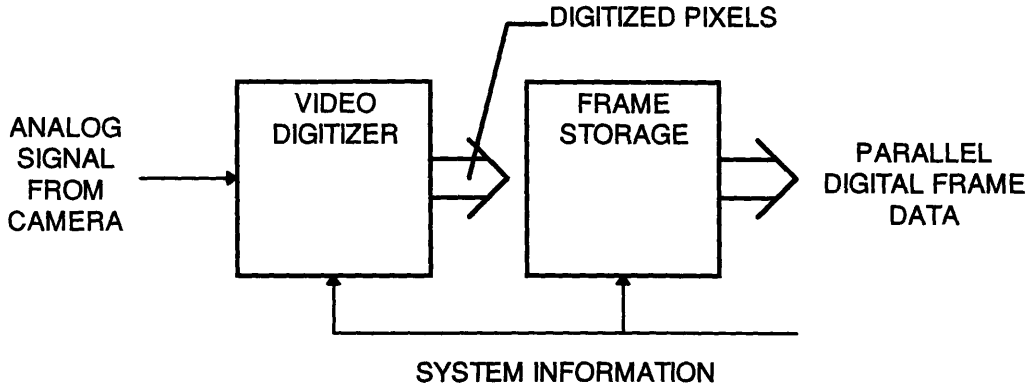


Figure 2.4 Block Diagram of the Image Storage Module

The frame storage module is used to convert the environment imaging signal into digital frame data and place it in a storage medium, see Figure 2.4. In our case, since we will be using a CCD camera to image the environment, it must convert an analog signal into digital pixel data that can be stored easily. The standard terminology for a device that converts camera output into pixel data is a "Frame Grabber."

As discussed in the Environment Imaging section, there are different video signal standards for the output of a CCD camera. Most often a Frame Grabber is designed to accept more than one video signal standard; a typical Frame Grabber will automatically adjust to the standard of the video signal connected to it.

Another important feature of some Frame Grabbers is the ability to store the pixel data it has created. The storage media can be of any type; some of the different types of media available are compact disc (CD), tape, hard disk, and non-volatile/ volatile Random Access Memory (RAM). The type of storage media dictates where the preprocessing module will have to look for the digital image information. The key factors in choosing the storage media are the amount of collected data, which is determined by the conversion rate of the Frame Grabber, and the ability to process the image data.

The conversion rate of a Frame Grabber, the speed that it can convert new analog signal data into digital pixel data, is called the "frame rate." The units of a frame rate are frames per second, which is the same units as the output of a CCD camera. The typical frame rate output for a CCD camera is 30 frames per second; this is actually predetermined by the video standard being used. It has been determined that at 30 frames per second the human eye cannot detect the discreet change between any pair of frames.

The Frame Grabber frame rate is directly proportional to the number of pixels it generates from each video frame produced by the camera; the more pixels it produces from a video frame the higher the resolution of the converted image. The number of digital pixels that a Frame Grabber produces can be more or less than the number of CCD pixels in the camera used to create the analog video signal. A frame grabber that produces the same number of digitized pixels as the number of CCD pixels in the camera is sometimes referred to as a matched system.

The last important characteristic of a Frame Grabber is the amount of power it consumes. A typical Frame Grabber consumes about 3 to 4 Watts.

In our application for autonomous systems we need a Frame Storage Module that has a high frame rate with storage capabilities. The frame rate should be as high as possible and the memory needs to be RAM, which will allow the fastest access to it. The power consumption of the Frame Grabber needs to be as low as possible. Today, there are commercially available Frame Grabbers that can act as the complete Frame Storage Module.

The Preprocessing Module

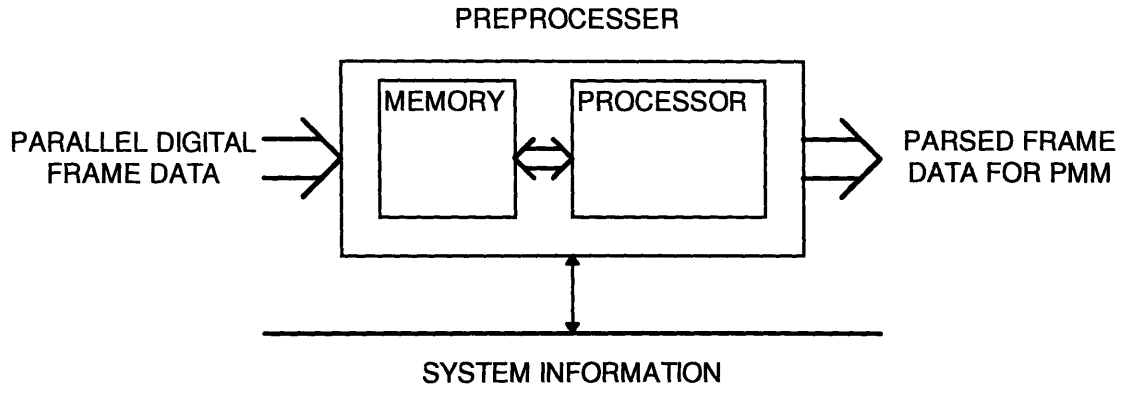


Figure 2.5 Block Diagram of the Preprocessing Module

The preprocessing module is used as a pixel distribution system. It removes the digital frame data stored by the Frame Storage Module and performs any filtering, converting, and parsing necessary to make the data suitable for the Pattern Matching Module, see Figure 2.5.

The processing system does not have to be very complex; it must have a substantial amount of memory and be fast enough to keep up with the Frame Storage Module. The amount of memory must be sufficient to hold all of the pixel data for two complete frames and all of the associated buffers. All of the pixel data extracted by the Preprocessor is in single bytes, thus any filtering and converting will not involve floating point computations (or floating point processor.) The last function of the Preprocessor is parsing. The digital frame data will be broken up into smaller blocks that the Pattern Matching Module (PMM) will process. The Preprocessor must be able to easily send data to the PMM and keep track of its status in order to make the most efficient use of the Preprocessor to PMM pipeline.

In our application there are many suitable microprocessors and hardware that could easily serve as the Preprocessor. The most important feature is it's memory access and data transfer speeds.

The Pattern Matching Module

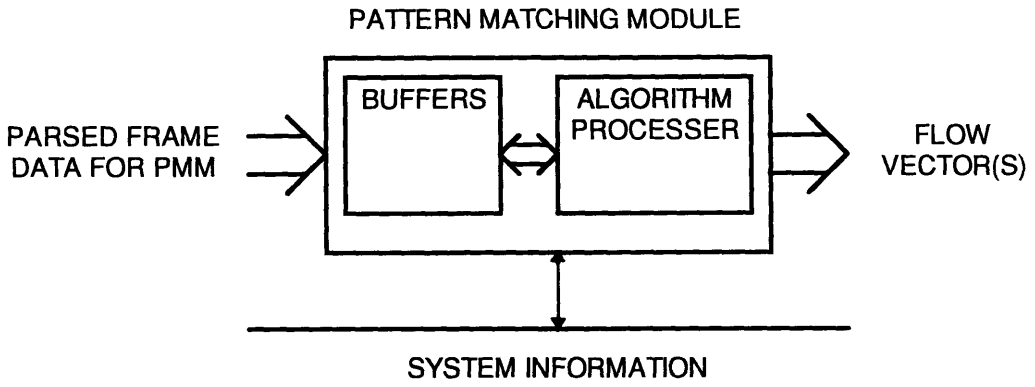


Figure 2.6 Block Diagram of the Pattern Matching Module

The Pattern Matching Module (PMM) finds a data block from one frame in a search window from another frame, see Figure 2.6. The algorithm used to find the new location of the data block is given in Equation 1.15. If the size of the data block is kept relatively small the algorithm does not need any floating point processing support.

The PMM needs to be able to calculate the algorithm fast enough to keep up with the ability of the Preprocessor and Frame Storage Module. The shear number of calculations necessary in the algorithm and the frequency that the algorithm must be used tell us that a dedicated piece of hardware must be used as the PMM, and it must be very fast.

The PMM would be loaded with a search window and a data block; it could compute the best match location without any extra support. This is a key point, viewing the PMM as a black box allows us to use many PMM's in various configurations. Obviously, an important ability of the PMM is to combine more than one PMM in parallel.

All of these issues apply to our application. We want a fast stand alone PMM with the ability to be easily integrated into a parallel system.

The Postprocessing Module

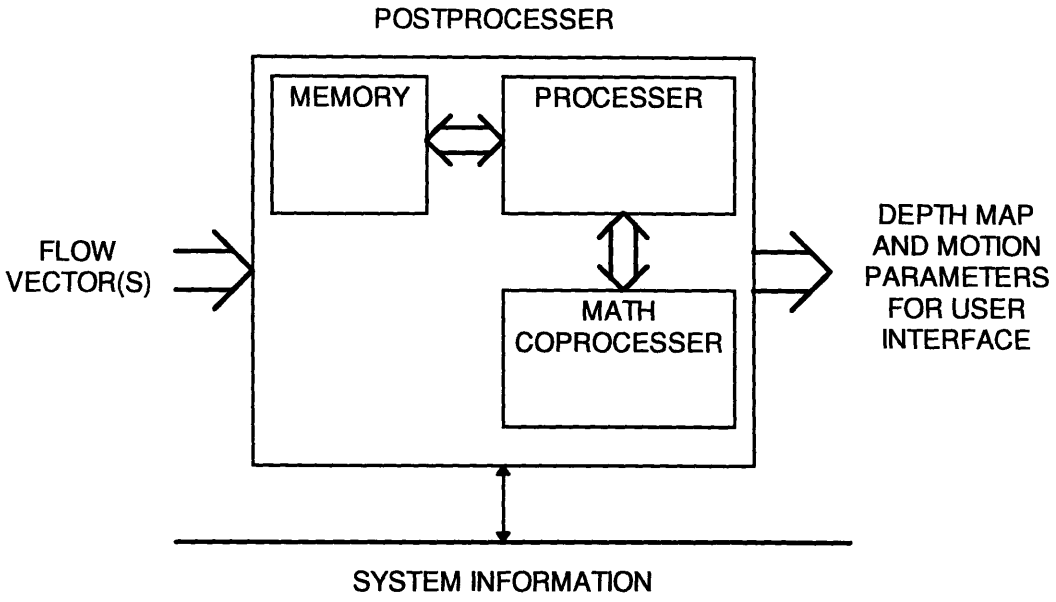


Figure 2.7 Block Diagram of the Postprocessing Module

The Postprocessing Module will collect all of the data from the PMM(s) and compute the six motion parameters, see Figure 2.7.

The motion parameter calculation, Equation 1.24, involve a large number of calculations; it will need floating point ability as well as the ability to handle large matrix manipulations. The Postprocessor, like the Preprocessor, must be constantly aware of the PMM(s) state in order to make good use of the PMM to Postprocessing gate. Equally important, the Postprocessor must have an efficient way to receive data from the PMM(s).

The PMM calculations and the Postprocessing form the largest computational bottlenecks for the Vision System.

In our application the Postprocessor is very similar to the Preprocessor, except that it must be able to process large mathematical computations quickly, and does not require as much data memory.

The User Interface

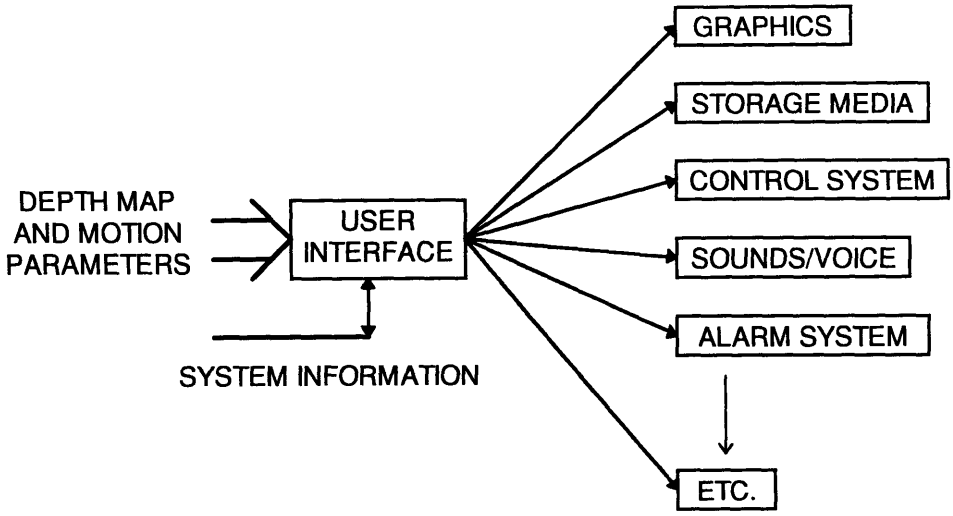


Figure 2.8 The Infinite Interface

The User Interface would receive the digital information generated about the motion parameters and the obstacle depth map and present them to the system requiring the results. In essence, this is the overall strength on this vision system, the interface can have many different forms depending on the functional need.

The most common use for the system will be as a navigational aid to an autonomous system. The autonomous system could be a sophisticated robotic platform or possibly a blind person; in either case a visual interface is not necessary. The robotic platform would merely like the digital data reflecting the motion parameters as well as the depth map. A blind person, however, may require a tone in each ear indicating the relative proximity to the nearest obstacles, and perhaps a supplementary voice for any additional information.

A visual interface is much more challenging because of the way that this information can be conveyed to a person viewing it. One useful way would be to generate a three dimensional obstacle map in which a simulation of the system using the vision system could be placed.

The Prototype Vision System Architecture

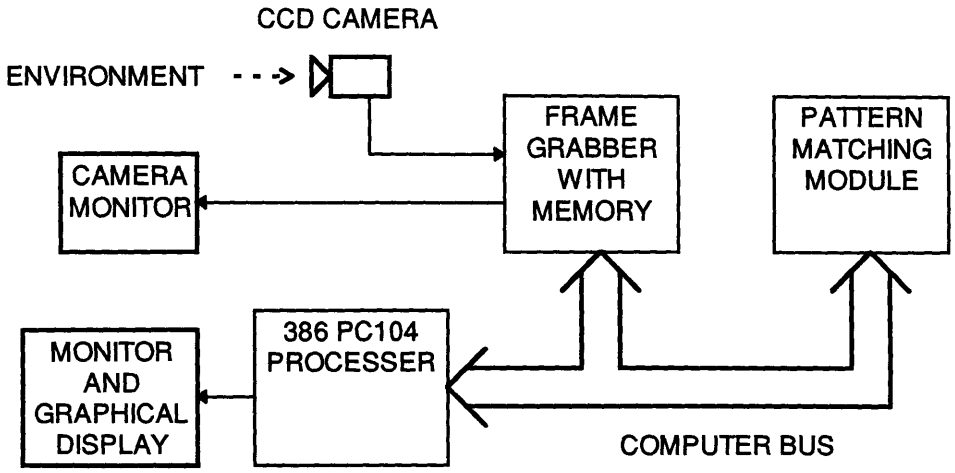


Figure 2.9 The Prototype Vision System Architecture

After considering all of the possibilities involved in constructing a Vision System a proof of concept system was constructed, see Figure 2.9.

It is a monocular vision system that has a single black and white CCD camera, frame grabber, pattern matching module, and processor. The frame grabber and PMM are at different address locations on the processor's bus; this simple architecture was used to develop the basic hardware and software necessary for a completely functional vision system.

The Vision System functions in the following way. The camera is operating constantly, converting the environment into a video signal. The Frame Grabber, on command from the processor, grabs two frames and stores them in memory. The processor then removes, filters, and parses each frame into its own memory. The parsed frames are fed through the PMM in datablock - search window pairs and the results are stored into the processors memory. After all of the flow vectors have been stored the processor runs the motion algorithm in Figure 1.6 to calculation the motion parameters and the depth map. Once the depth map and motion parameters have been calculated they can be used in any imaginable way.

In terms of the functional components listed in Figure 2.1, the PC104 processor serves as the Preprocessing, Postprocessing, and User Interface Modules. It is a very simple system that can be made quickly and robustly.

The ultimate strength in using this architecture for the prototype system is that all of the pieces of the system can be simulated in software before they are constructed in hardware; this is made easy because we have a centralized single processor system The simulation flexibility also allows the developer to test alternate hardware architectures and troubleshoot both the hardware and software more easily.

Chapter Summary

The chapter began by discussing the natural image processing flow of a general Vision System. The functional components of the natural flow are the Environment Imager, Image Storage Module, Preprocessor, Pattern Matching Module, Postprocessor, and User Interface.

The Environment Imager is used to convert an optical image of the environment into an analog signal. A typical environment imager is a black and white CCD camera. The most important considerations in using a CCD camera for imaging are the shape of the cells in the CCD array, the resolution of the CCD, the ability to control the amount of light the enters the camera, and the ability to improve the image contrast.

The Image Storage Module is used to digitize the analog signal from the Environment Imager and store the information. The design issues involved with Image Storage Module are the type of storage medium, frame or conversion rate, and the generated pixel resolution.

The Preprocessing Module retrieves, filters, and parses the pixel data created by the Image Storage Module. The Preprocessor needs to have a significant amount of memory and speed. It does not have to support floating point calculations.

The Pattern Matching Module is used to generate flow vectors that will make up our motion field. It performs this function by finding a given pattern, called a datablock, in a specific range, called a search window. The location of the datablock in the search window is the flow vector. The primary concerns about the PMM are that it can process quickly and act as a stand alone unit. The black box approach to the PMM will allow it to be used in a parallel architecture.

The Postprocessing Module will calculate the motion parameters. The Postprocessor needs to support floating point and matrix computations.

The User Interface is the most elegant part of the whole system. It can be tailored for any application where there is a need for navigational information. One example as that the User Interface could supply information to the navigational control system of an autonomous vehicle.

The end of the chapter is a brief introduction to the prototype Vision System constructed for this thesis. The prototype architecture is the simplest way to implement this proof of concept Vision System. It is a monocular Vision System with a single camera, Frame Grabber, Pattern Matching Module, and Processor. The single processor performs the Preprocessing, Postprocessing, and User Interface functions. The camera, of course, serves as the Environment Imager. The Frame Grabber, with internal memory, performs the function of the Image Storage Module, and the Pattern Matching Module is itself. The prototype architecture is easy to develop because every component can be simulated in software before being built. The ability to accurately model each component in software also helps troubleshoot the hardware being developed as well as test different hardware architectures.

CHAPTER THREE
THE PROTOTYPE VISION SYSTEM HARDWARE

This chapter contains all of the specific details in the design and construction of the prototype Vision System hardware described in chapters one and two. The system is discussed in two parts, hardware and software; this chapter is dedicated to the hardware development. Both the hardware and the software chapters adhere to the same progression that was used in chapter two, where design decisions of each component are presented sequentially. Figure 2.9 is included again as Figure 3.1 for reference. The intent of this chapter is to provide enough information so that the reader will have a complete understanding of the prototype system hardware.

It should be stressed that any single block in Figure 3.1 has to be designed, or specified, in conjunction with all of the blocks that it interfaces with. The system design issues will be individually addressed for each component.

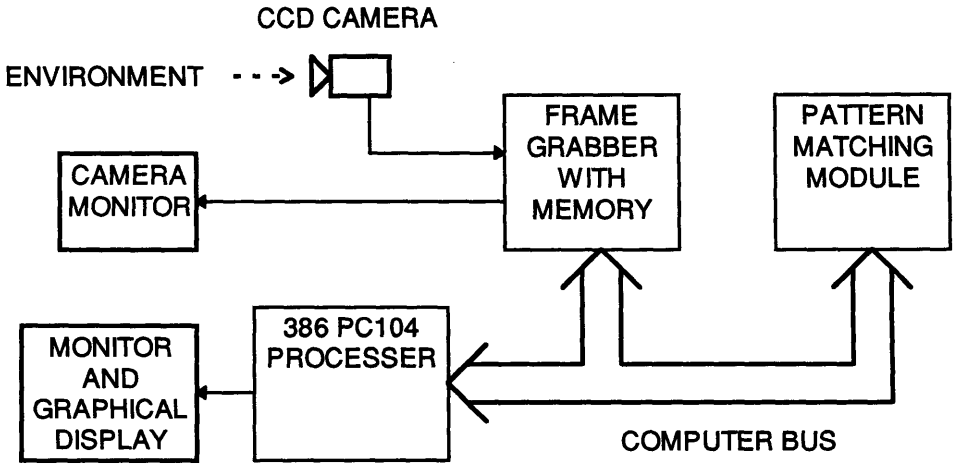


Figure 3.1 The Prototype Vision System Architecture

3.1 The Prototype Hardware

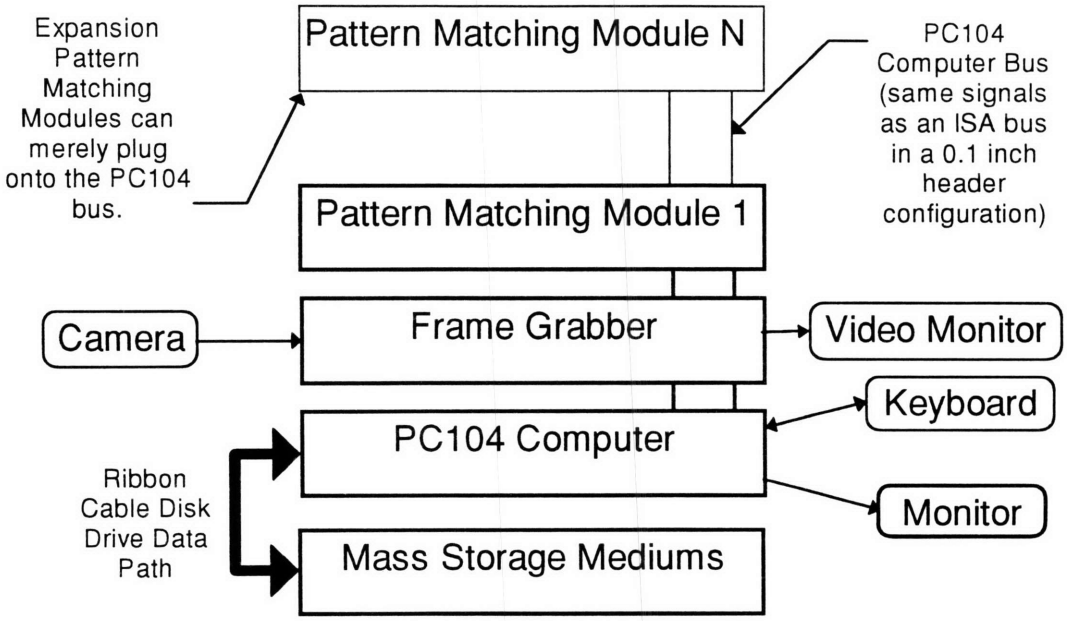


Figure 3.2 Conceptual Hardware Block Diagram

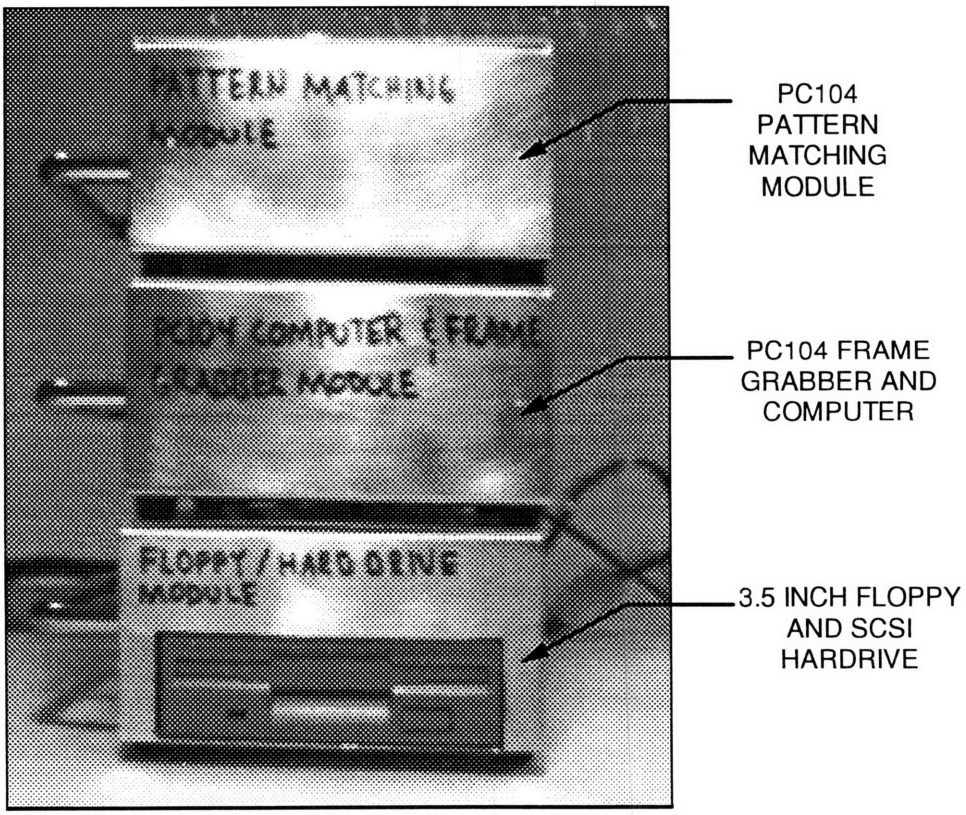


Figure 3.3 The Prototype Vision System Hardware

3.2 CCD Camera

The CCD camera is used as the Environment Imager. The camera that was chosen is a PULNiX TM7-CN. (See Figure 3.4) The TM7-CN can be used as a normal video camera, but is tailored for machine vision systems.

Figure 3.4 Pulnix Black and White CCD Camera

Component Specifications and Features

- 768 horizontal pixels by 494 vertical pixels
- Camera is for Black and White applications
- Rectangular CCD cells 8.4μm horizontal by 9.8μm vertical
- Camera provides Auto Iris feature for lens
- Camera has Gamma contrast control
- CCD is 0.5 inches square overall
- Very small package
- CCD is sensitive to 1 LUX

Design Reasoning

The reasoning for the selection of the camera is based around what we need from our environment. Cameras are instantly divided into two camps -black and white,

sometimes called monochrome, and color. In reality, our machine vision system is only observing the gross movements of the camera through its' surroundings; this motion can be measured with a basic black & white camera. The computational overhead to process color is exactly three times as much -this alone makes it prohibitive since we are aiming for a fast stand alone vision processing system. The next issue is that of image resolution.

The number of horizontal and vertical pixels of the PULNiX TM7 is ample enough for the proposed camera and Frame Grabber system; the Frame Grabber, which has not been discussed in detail yet, has a lower resolution on its best setting and thus will not be "creating" pixels from wholly interpolated data. This is a strange point. The issue is that the camera creates an analog signal from its discreet number of CCD cells. The analog signal produced by the camera from discreet data is intentionally, and even unintentionally by loading, smeared to make it appear smooth. The Frame Grabber can produce any number of pixels from the smeared video signal since it is analog. It makes sense, however, that the Frame Grabber should not try to extract more pixels from the video signal than were used to create it. In our system the resolution of the video camera is sufficient and a more expensive camera would not yield better performance.

As was mentioned in Chapter Two, it is convenient to choose a black and white camera with a simple pixel geometry, auto iris, and contrast control. The easiest pixel geometry to work with is square pixels which for most applications is the best choice. The TM7 does not have exactly square pixels even though it does contain a 0.5 inch square CCD. The TM-7 has 768 horizontal pixels by 494 vertical pixels. The auto iris feature controls the amount of light that hits the CCD. The camera's electronics measure the overall pixel intensity and open and close the lens iris to make full use of the CCD's dynamic range, down to 1 LUX. The last technical point is the ability to increase the camera contrast; the TM7 does this through a Gamma control that increases or decreases the contrast. Finally, the TM7 comes in a very small package that could be used for almost any autonomous application.

3.3 Auto Iris Lens

The lens that was chosen is a Computar MCA0813APC.(See Figure 3.5) Computar lenses are fairly popular for black and white camera applications. Often, they are used on the surveillance cameras at Automatic Teller Machines (ATM).

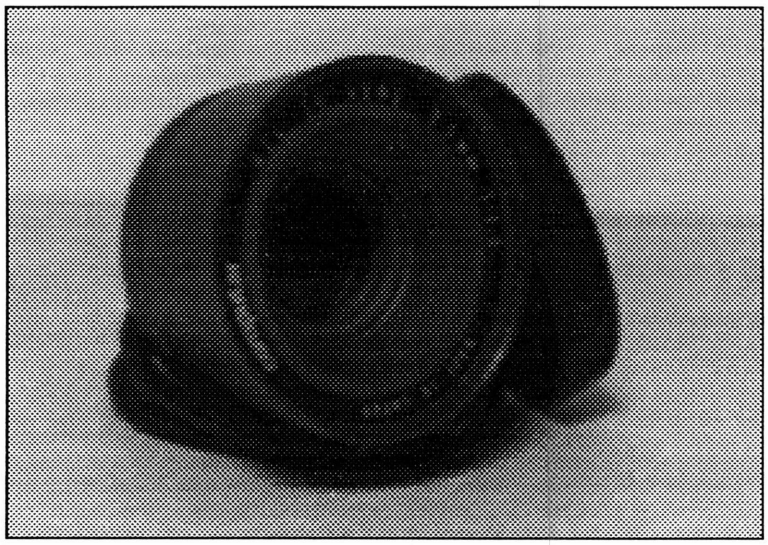


Figure 3.5 Computar Auto Iris Lens

Component Specifications and Features

- Auto Iris for pixel light metering control
- 8.5 millimeter focal (i.e. wide angle view lens)
- Adjustable video signal level
- Auto Close on power down

Design Reasoning

The reasoning here is very direct. The most important features are the Auto Iris and focal length.

The Auto Iris feature gives the camera the ability to react to changing light conditions the way the human eye does. Our pattern matching method depends upon matching pixel brightness patterns. Since we expect to eventually use the system on a robotic platform in environments with varying light conditions, we need to be able to maintain, or slowly vary, the relative brightness of the pixels on our CCD. If our robotic

platform moved from a dark region to a bright area, in order to maintain the brightness of the image we need to close or open the iris by varying amounts. Analogously, a human eye will increase or reduce the size of its pupil depending upon whether it is receiving too much or too little light; it does this by constricting and relaxing the human eye iris. The Auto Iris in our lens provides our machine vision system with a similar feedback iris control. The lens contains electronics that try to maintain the same average pixel brightness as the ambient light of the environment changes. Without the Auto Iris feature our system would be less stable since the data we would receive from the camera and hence the frame grabbing module would be changing too quickly. We assume that we will be grabbing and processing frames more frequently than the auto iris is changing. The electronics used in the Auto Iris control also close the lens during a power loss.

The focal length is the next most important characteristic. In order to properly implement the equations of chapter one we need to know our focal length exactly since we use it to normalize our CCD x & y locations. The ability to normalize, however, is not the only importance of the focal length. The focal length is indicative of the angular view of the lens -the smaller the focal length the larger the angular view. In a machine vision application it is important to have a very wide angle lens -the more information and motion you can capture with your lens/camera combination the better. On the other hand, the more we increase the angular view of the lens the more distorted and non-linear the image appears at its edges. Our approach depends upon as much linearity as possible, so the lens we chose has a 62 degree angle of view.

The adjustable video level merely makes the whole video image seem brighter or darker. In our system, as long as we have a fair amount of contrast and don't allow the video level to wash out the image details it is unimportant. The only constraint is that it must remain constant during the operation of the system; this idea follows the same bearing as the need for Auto Iris. We would not want to have significant changes in the quality and contrast of our images between iterations of our algorithm.

3.4 Frame Grabber With Memory

The Frame Grabber, see figure 3.6, is used as the complete Image Storage Module. It converts two analog frames into digital pixel data and stores them into RAM. Many of the characteristics of the Frame Grabber must match those of the CCD camera. The Frame Grabber that was chosen was a ImageNation CX104 with overlay RAM.

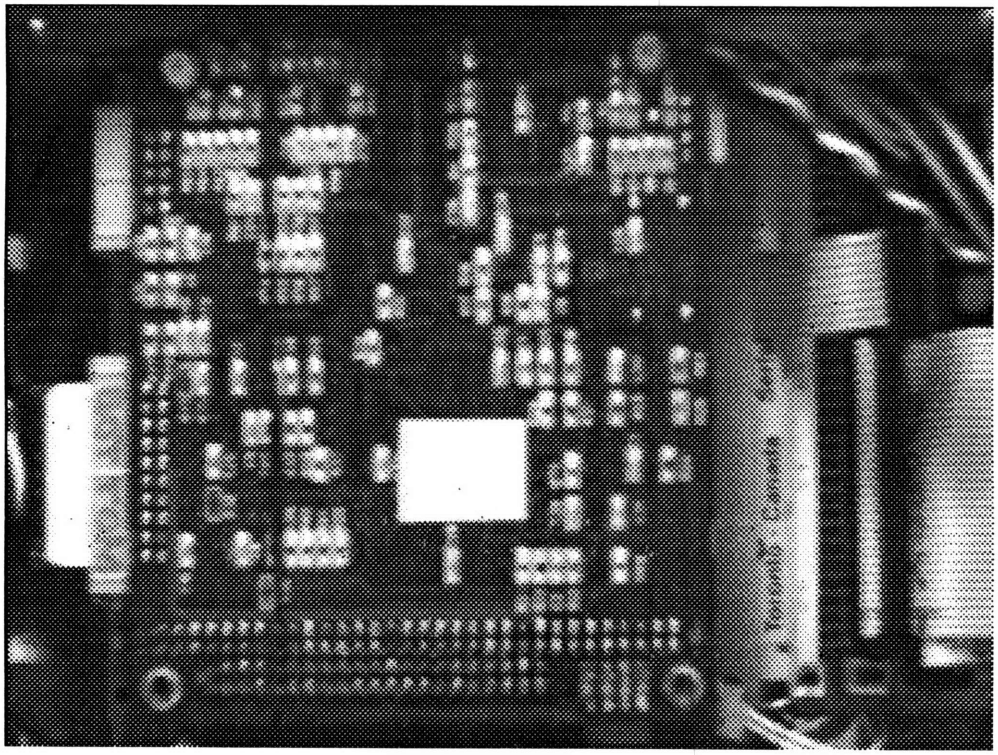


Figure 3.6 ImageNation Frame Grabber

Component Specifications and Features

- 8 bit pixel resolution
- Utilizes a Black and White video camera
- Accepts CCIR/PAL and NTSC video formats
- High and Low resolution modes
- Internal RAM for image storage
- Overlay RAM for external data buffer and display
- Separate camera monitor output
- Plugs into a PC104 computer bus
- Complete C library with some source code

Design Reasoning

The Frame Grabber must be designed for black and white, or monochrome, video. The Frame Grabber should be of a lower resolution than the camera. The CX104 has two resolution modes, high and low resolution. In high resolution the CX104 generates a frame of 512 horizontal pixels by 486 vertical pixels, and in low resolution it generates a frame of 256 horizontal pixels by 243 vertical pixels. Every pixel the CX104 creates is an 8 bit, or byte, value; this allows each pixel to be 256 shades of gray. The height and width of these actual pixels generated by the Frame Grabber depend upon the combination of the camera/Frame Grabber; this is because the camera creates a smeared signal of its pixels from a wide angle lens at about 30 frames/second and the Frame Grabber interprets the signal with its own sampling clock. The translation of the real world coordinates onto the "virtual" pixels created by our system is discussed in camera/Frame Grabber calibration section. The analytical deduction of the width of the virtual pixels would be quite challenging, since it would be difficult to make accurate measurements. As long as we use the maximum resolution of the Frame Grabber in conjunction with the camera, the virtual height of the pixel is known to be its real height; however, in this application the frame grabber is used in its low resolution mode which combines vertical lines of video data as well as consecutive horizontal pixels, therefore, we also have to compute a "virutal" pixel height as well.

Another important aspect of the CX104 is its local memory and the way that it can be accessed. The CX104 is capable of storing four low resolution images or a single high resolution image. It has a complete C library, including source code, that allows one to perform operations on the images, display data supplied by the user, as well as access its memory. In the prototype system the Grabber is used as a buffer where the images are temporarily stored before being parsed into digestible pieces for the Pattern Matching Module by the PC104 computer. The existence of the C library, which contains image data exchanging functions, assures us that we will be able to quickly integrate the Frame Grabber into the system.

The CX104 is a PC104 form factor device. The PC104 form factor, as was mentioned in Chapter 2, is a compact computer standard. It is the best choice for variable

applications because of its small size and manageable power requirements. Each card of a PC104 computer is 16 square inches (4 inches on a side) and has a bus architecture that is very similar to ISA. The computer cards stack on top of one another with each card performing a specific function. Thus, the Frame Grabber is a single addressable card on the main computer bus and, of course, has its own interrupt. The PC104 bus is used as the data transfer medium for all digital operations internal to the Vision System.

3.5 The PC104 Computer

The PC104 computer will serve as the Preprocessor, Postprocessor, and User Interface. The Preprocessing function filters and parses the digital frame data supplied by the Frame Grabber. The Postprocessor function collects and stores all of the Pattern Matching Results, which is the motion field, and calculates both the six motion parameters and the depth map. The User Interface function is tailored to the host system which is using the Vision System; it supplies the host system with whatever information it wants and in whatever form it needs it. The PC104 computer that was chosen for the prototype system was a Megatel 386 with 10Mb of DRAM.

Component Specifications and Features

- PC104 form factor
- 10 Mb of DRAM
- Capable of accepting a Math Coprocessor
- On board Ethernet
- On board VGA driving hardware
- Supports floppy and SCSI hard drives
- 2Mb of FLASH solid state disc
- 386 processor that runs at 25Mhz
- Has standard IBM addressing techniques
- Supports DOS, Windows, Borland C

Design Reasoning

Building on the previous components, the main computer is a PC104 form factor device. Despite the small size of a PC104 device, the Megatel computer offered a lot of convenient features that make it attractive for prototype development. All of the features

listed above exist on a single PC104 surface mount PCB. The most important features are the large amount of available DRAM, the VGA driver, the floppy and hard drive supports, and the 2Mb of solid state disc.

The DRAM ensures that we will have fast memory access and enough memory to both store digital frame data, the associated buffers, and the matrices necessary for motion calculation. The VGA driving ability makes this computer ideal as a User Interface; the VGA driver can be used to help develop the system and as an additional capability of the completed prototype. The need for drive support is very straightforward, we need a way to store large amounts of information and development software. The most odd feature of the whole computer is the solid state disc; this disc will be used as the boot drive and for storing all of the compiled vision software. After the prototype is complete the hard and floppy drives can be permanently removed so that the Vision System can act as a stand alone or embedded unit.

Typically, PC104 products lag behind conventional products by about one to two years. At the time of the work done on this thesis was started only a 25Mhz computer was available with all of the above features. 25Mhz, however, is fast enough for prototyping purposes. Many of the other on board features, like the on board Ethernet, were not used.

Support Devices

- Maxtor 256Mb SCSI hard drive
- Samsung 3.5 floppy drive
- IBM VGA monitor

3.6 The Pattern Matching Module

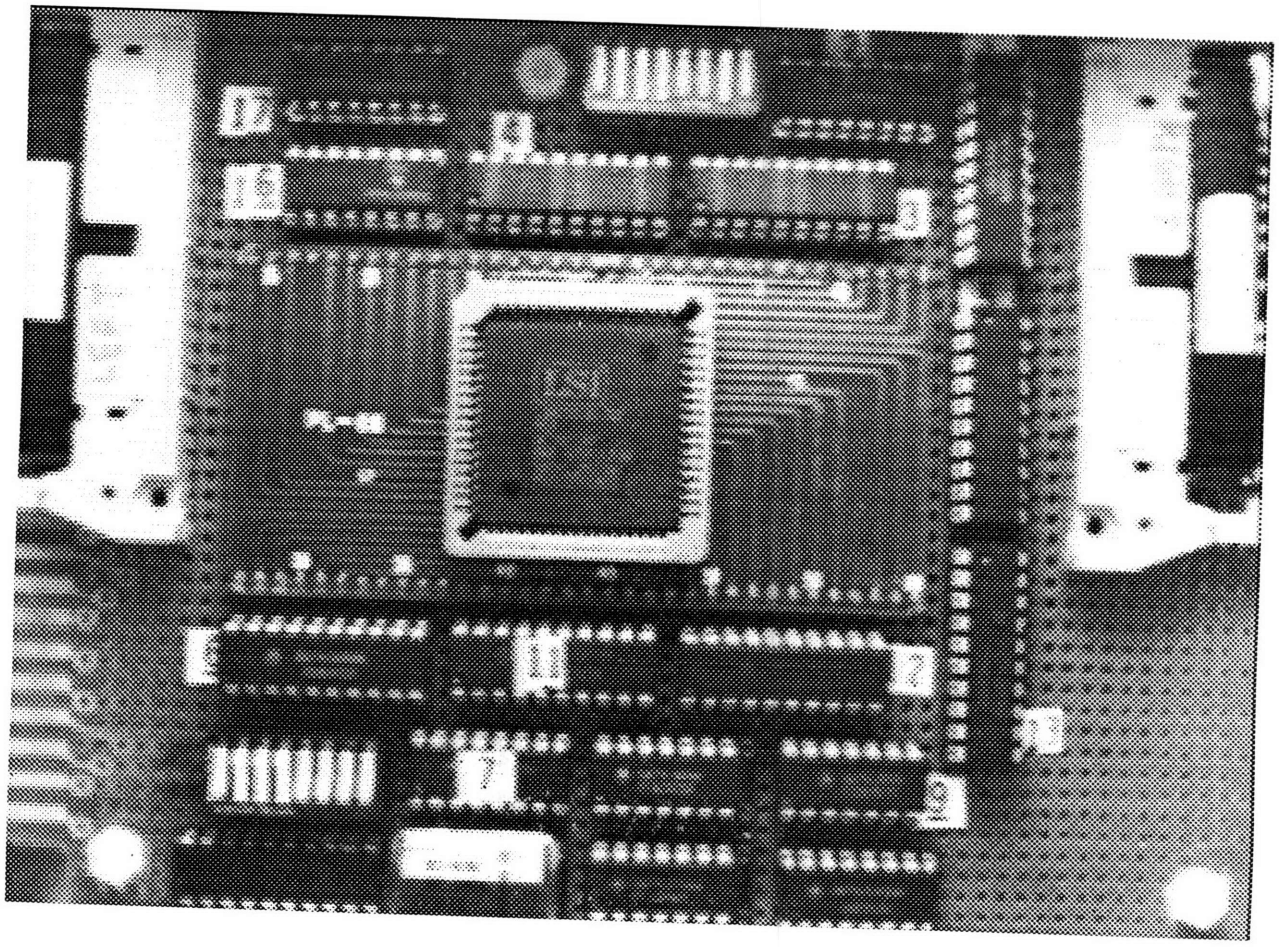


Figure 3.7 The Prototype Pattern Matching Module

The Pattern Matching Module (PMM), see figure 3.7, is the last piece of hardware that will be discussed within this thesis; it is truly the heart of the work accomplished. The PMM is used to find a piece of one image in a second image resulting in a flow vector. After all of the comparisons between two frames have been performed, the flow vectors are assumed to be a measure of the flow field. The PMM performs the bulk of the mathematical computations that are necessary to generate the motion flow field. The PMM that we need in our architecture and framework did not exist- it had to be designed and built.

Design Considerations

The overall need for the prototype system is clear -proof of concept. Our prototype will be the first cut at a functional Pattern Matching Module that can easily be adapted to more sophisticated systems. Therefore, we need a simple design that is reasonably fast and both easy to develop, construct and understand. Simplicity, of course, is never easy to achieve. We can, however, minimize our overhead. The following precepts were decided upon for the PMM:

- Use a standard C compiler.
- Develop hardware around existing C functions.
- Develop addressable hardware that plugs onto the computer bus.
- Plan for expandability where possible.
- Use as many "off the shelf" components as possible.

On the issue of software development we can facilitate programming by mandating that our hardware should conform to existing functions in a standard C compiler and avoid programming any drivers lower than these. Further, we can obtain a minimum level of expandability by making our hardware addressable since we could then have more than one PMM, as indicated in figure 3.2. The use of "off the shelf" components is merely a pragmatic issue.

3.6.1 The Motion Estimation Processor

The center of attention in figure 3.7 is the large square 68 pin chip in a Plastic Leaded Chip Carrier (PLCC) that is the LSI Logic L64720 Motion Estimation Processor (MEP). Stephen Lynn, in his thesis entitled "A Motion Vision System for a Martian Micro-Rover" suggests the use of the LSI L64720 MEP to create a motion flow field; this is such a good idea that it is what we attempt to do here! The L64720 MEP was originally designed for the High Definition Television (HDTV) market and is hardwired to run the algorithm described by equation 1.15; this is the method by which we create a each flow vector in our flow field. The MEP has an array of 32 processors and can run at speeds up

to 40Mhz. It can process video data in real time, that is at 30 frames per second. It is the heart of the PMM and utilizing it substantially reduces the complexity of the pattern matching hardware. The selection of the MEP means that the rest of the hardware design is in the glue logic that makes it pluggable onto the computer bus.

3.6.2 Addressing and Decoding Logic

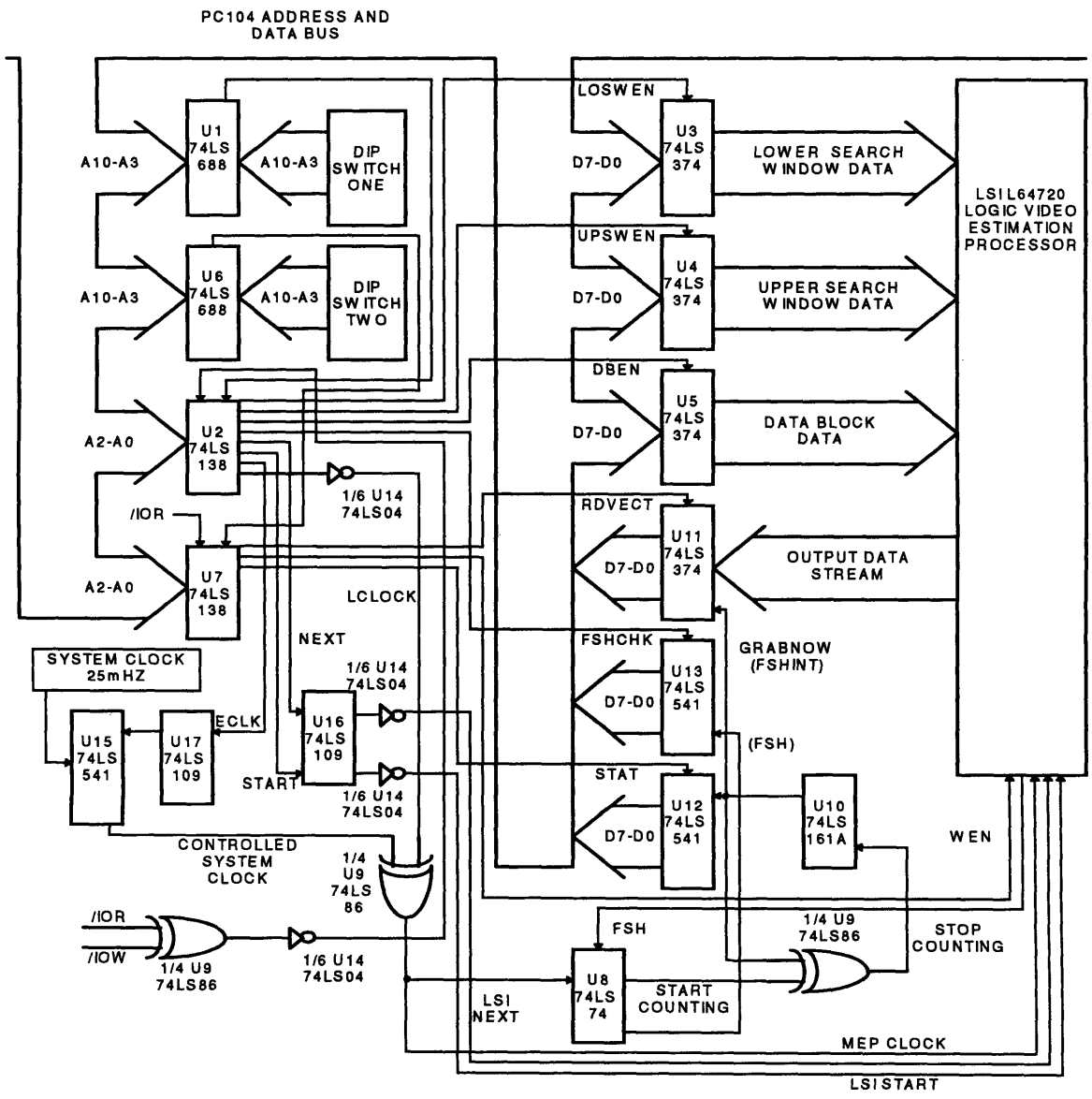


Figure 3.8 Functional Block Diagram of PMM

The addressing and decoding logic in figure 3.8 is used to allow a system to be connected onto a bus. The logic will watch the addresses that the computer is requesting service with and when the logic's address is present it enables the correct hardware to function. The enabled hardware may then either write or read a byte to or from the bus, or perform a specific function. It should be noted that there is a full schematic of the PMM in Appendix A.

For our design we will not be using custom software, but rather the C functions outportb and inportb. These two functions place an address on the bus, and then read or write a byte -they are based upon the standard International Business Machine (IBM) Personal Computer (PC) Input/Output cycle in figure 3.9. The byte being read or written follows the I/O address, and it is latched on the rising edge of the /IOR or /IOW.

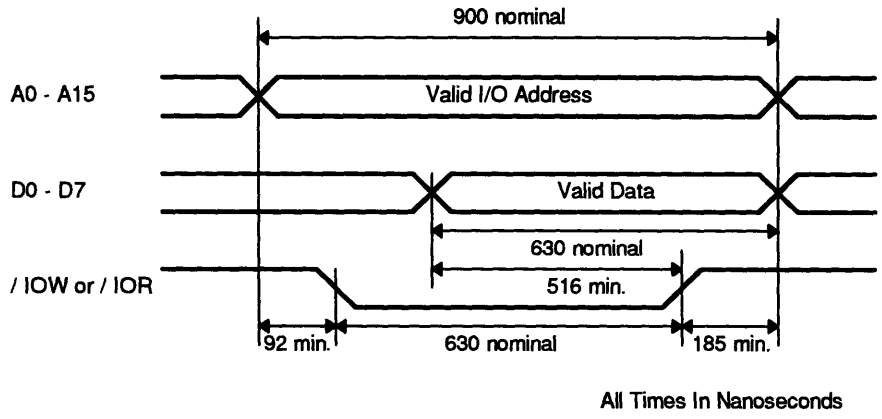


Figure 3.9 IBM PC I/O Cycles

First, since we are intending to use canned logic, like our MEP, we should decide upon a logic family. TTL families are traditionally a good choice for prototyping because they can take a few bench top mistakes before they become damaged. The choice for this prototype is the fastest and yet lowest power of the TTL families -LSTTL, which stands for Low power Schottky Transistor Transistor Logic. After a few calculations from figure 3.9 it is obvious that these devices have small enough latencies that we should not have any problems.

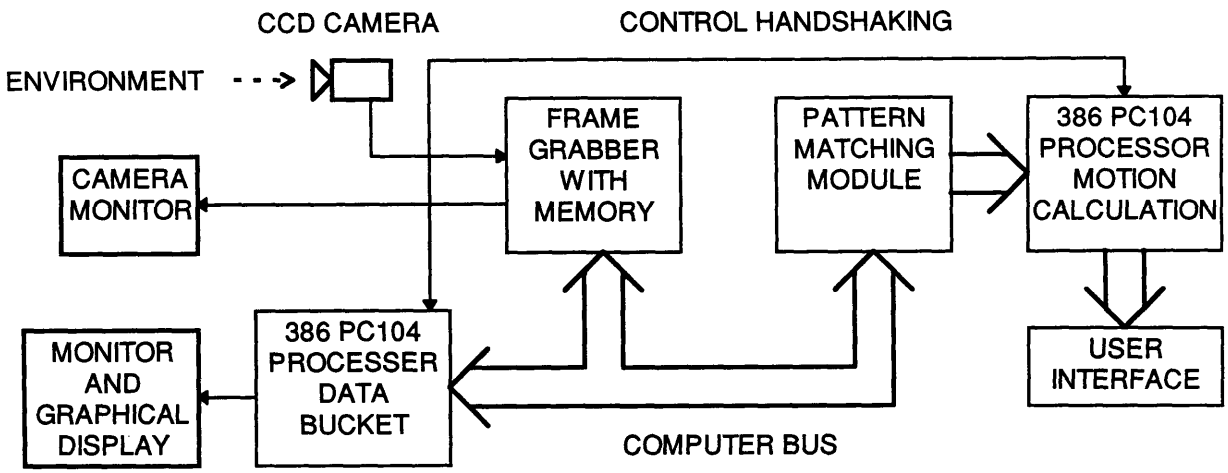


Figure 3.10 Original Prototype Concept

One of the original concepts for the vision system was a variation of figure 3.1, where we had two PC104 386 computers instead of one, see figure 3.10. Each processor would have a dedicated function.

The first processor would act as a data bucket. The frame Grabber would dump raw pixel data into the data bucket and the first processor in conjunction with the second processor would feed it through the Pattern Matching Module until the second processor got a full flow field. Once the second processor got a full flow field it would calculate and return the user interface. The original concept is present in the way the actual addressing/decoding hardware is implemented -there are two sets of addressing/ decoding hardware that were originally for the two separate computer buses of figure 3.10. The adaptability from a second bus is observable in figure 3.8 because we have two 74LS688's, U1 and U6, and two 74LS541's, U12 and U13, used as status registers.

The easiest way to do the addressing logic is by using 74LS688's which are eight bit magnitude comparators, U1 and U6 on figure 3.8. These comparators have sixteen inputs, which are two eight bit ports, and one output. We can place eight bits of our PMM address, A10-A3, on one of the eight bit ports using a Dual row In Line (DIP) switch and let the other port snoop the eight address lines, A10-A3, of the bus. When the two ports have identical bytes the comparator outputs a low. The lower three address lines, A2-A0

are left for our decoding logic. In essence, the address A10-A0, with the first three bits set to zero is a base address. In our current system of figure 3.8 we have a base address for each 74LS688.

The simplest way to do decoding is by using 74LS138's. The 138 is a three to eight line decoder, U2 and U7 on figure 3.8. These decoders have three input pins, three enables and eight active low outputs. The three inputs are connected to the three lowest address lines A0, A1, A2. This is so that we can decode the base address and the seven sequential addresses after it. The sequential addresses that the 74LS138's decode are used for latching our data bytes and asserting some control signals. Each 138 has to have three separate enables signals asserted before it will decode. One of the three enable lines is constantly enabled and another is fed directly from one of the 74LS688's. The last line is fed into either /IOW or /IOR at any given time; this ensures that the circuit only decodes addresses when it is reading or writing. Another important use for the /IOW and /IOR comes during latching.

3.6.3 Latches and Drivers

The latches and drivers in are used to access read from and write to the eight bit data bus. We use 74LS374's for latches and 74LS541's as line drivers in figure 3.8. Both of these devices are tri-state logic suitable for bus connections. A tri-state component can supply a high, a low, and appear as a nearly open circuit. It is important to use tri-state devices in this application because they have a high current fan out and yet do not impose a significant strain on other bus circuits. Again, as with the addressing and decoding logic some of the circuitry of the final layout is intended for the full two processor implementation of figure 3.10.

The 74LS374's, U3, U4, U5, U11, serve as our port registers for the MEP. One is used for each eight bit port of the MEP of which there are four. Three of the four ports on the MEP are used as data inputs for its internal search and data window buffers. The last port is where the output flow vector byte is taken. U3 and U4 are for search window data,

U5 is for the data block bytes, and U11 captures the output flow vector; this terminology is defined at the end of chapter one.

The three 74LS541's that are used are for placing status bytes on the PC104 computer data bus and for disabling and enabling the MEP run time clock. The first driver, U13, is used to indicate when the MEP has finished calculating a flow vector, the second, U12, indicates when the external logic has latched the correct byte from the MEP output stream. The last driver, U15, is used to disable and enable a 25Mhz system clock for the fast operation of the MEP; this is not a typical nor recommended practice. In this application great care was taken to insure that the circuit would not receive any interference from the 25Mhz clock. The MEP is meant to be used as a completely embedded device with it's own supporting memory. In order to develop this system in a timely fashion, the memory system of the PC104 computer has been used; it is therefore necessary to disable the actual MEP clock in order to load it's memory over the computer bus -which is infinitely slow. The line driver "switch" is merely a convenience for this prototype, not a practical solution.

3.6.4 Control Logic

As one might imagine control logic is everything else in figure 3.8. The control logic is composed of a 74LS04, 74LS74, 74LS86 74LS161A, and two 74LS109's. The 74LS109 is a dual J-K flip flop. The 74LS04 is a bank of six inverters. The 74LS74 is a dual D latch. The 74LS86 is a quad EX OR. The 74LS161 is an asynchronously resettable counter. The function of every device in and of itself does not tell the whole story. Here we are trying to understand the operation of each part. In chapter four we will look at the control software the whole Pattern Matching Module will come into focus.

The two 74LS109's are U16 and U17. The first J-K flip flop, U16, is used for clocking the data block and search window bytes into the MEP and for providing the signal for the MEP to start computing. The second J-K flip flop, U17, is used for enabling the run time 25Mhz clock to the MEP after it has been loaded and the signal to start has been initiated. All of these signals could be handled by an embedded programmable device

to increase the performance. Presently, each control function is given an address on the bus and are asserted by an function call in the software.

The 74LS04, of course, just inverts signals and is not of a great functional importance here. Likewise, the 74LS86 is just four EX OR gates and is used to keep certain signals from overlapping one another. For instance, the /IOR and /IOW that enable the 3 to 8 line decoder U2 are fed through one of the EX OR's to make sure that it is enabled only on exactly a read or exactly a write.

The 74LS74 provides a key control function. It receives the finish signal (FSH) from the MEP, sets the status byte of the U13 driver and activates the 74LS161A counter. The counter then counts the bytes that come out of the MEP and tells the latch, U11, when to capture the second byte of the output stream, which is the flow vector. The output stream of the MEP contains extra information about the positional errors after equation 1.15, the pattern matching algorithm, is executed. The byte capture function is important if we want to make any use of the 25Mhz clock, and hence the use of the MEP. This is because the computer cannot be made aware quickly enough that the MEP has finished calculating to grab the second byte of the MEP's automatic output stream.

3.7 Chapter Summary

The chapter started with a discussion of the conceptual hardware in figure 3.2. The conceptual hardware is implemented in figure 3.3. This vision system concept is a stand alone unit with all the elements and convenience of a normal desk top computer in a compact form. It is a bus based system that has the ability to be expanded, upgraded, and incorporated into a larger system. It uses all commercial components and the architecture of figure 3.1.

The first component from figure 3.2 discussed is the CCD camera. Today, there is nearly a different camera for every application. It is important to know exactly what kind of information is needed from the camera since it is usually the only sensor in a vision system. We chose the Pulnix TM7-CN which is a black and white camera with gamma

control, auto iris feedback control, and moderate CCD resolution. There is no need for a color camera in this application -it is too much information! Since we are performing pattern matching a black & white video signal carries all of the essential characteristics of the environment. The gamma control makes the images from the camera more crisp -it is a feature specifically for machine vision. Auto iris control gives the camera and lens combination the ability to compensate for varying lighting conditions independently. The number of pixels of the CCD is a system issue. There are very few Frame Grabbers for in the PC104 form factor. The camera resolution has to be larger than that of the Frame Grabber to have strong confidence in the images we grab, unless we have a matched system. The TM7-CN has nearly twice the resolution of the Frame Grabber.

The camera lens is as important as the camera, together they form our vision sensor. An attractive property of a vision system is the amount of data that can be grabbed in an instant. We want to grab as much data as possible without distortion around the image edges. A typical choice for such systems is a 60 degree viewing angle, which corresponds to a 8.5mm focal length. The lens should, of course, have supporting electronics and servomechanisms for auto iris light metering.

The Frame Grabber is completely prescribed by system parameters. Following our reasonable view about the camera, it should be a black & white grabber with a lower resolution than the camera. It should be in the PC104 form factor, have a complete C library for programming and internal image storage. The C library makes the unit useful to our already chosen programming language. The internal storage is necessary for any bus based system to operate efficiently.

The PC104 computer is the most interchangeable part of the system. As these computers get better we can always "plug" in a newer one later. Regardless of the exact manufacturer it is necessary to have a computer a large amount of RAM, at least 8Mb, and monitor and drive support.

The Pattern Matching Module is the largest part of the prototype design. The PMM stands as a proof of concept PC104 card. The PMM plugs right onto our computer

bus, and has its own address. If an application required more than one PMM they could merely be stacked as in figure 3.2. The PMM did not require any low level coding and was completely functional with existing C functions. For efficient design and implementation the PMM was completely designed from off the shelf components.

The most important feature of all of the prototype system hardware is that it works!

CHAPTER FOUR
THE PROTOTYPE VISION SYSTEM SOFTWARE

Software is a way of representing a process. This chapter will discuss all of the underlying processes that were implemented in software for controlling the Pattern Matching Module in figure 3.7, creating a flow field, and calculating motion. The chapter is organized as a spiral of abstractions. We quickly revisit the general Vision System abstraction and then focus on the processes and function calls in our specific implementation. The intent of the chapter is to supply enough of the fundamental components so that the reader can completely understand the main control loop, FTST13LA.CPP. Although this chapter will not contain all of the details used in the software, it will provide enough understanding so that the interested reader can unravel the rest of the code in Appendix B.

The language we will use to describe the processes is C. All of the code was developed using the Borland C++ 4.51 environment. The choice of the Borland C++ environment was not arbitrary; it is important to develop code using a framework with excellent debugging and compiling features; this can substantially reduce development time.

The Overall System Process

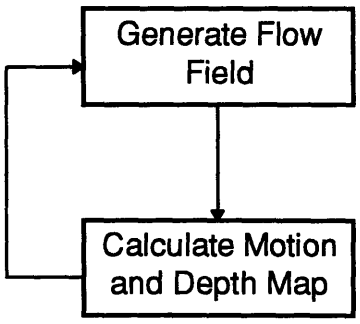


Figure 4.1 Two Step System Process

The overall system process is indicated in figure 4.1;it is a simple two step process. The first step is to generate a flow field and the second step is the iterative calculation of the motion and depth map. We would like to be able to perform this two step process as

quickly as possible. Generating the flow field uses both hardware and software while calculating motion and depth is strictly software.

It is important to note that the terms "flow field" and "motion field" are used interchangeably because in our model it is assumed that they are equivalent. A flow field is a two dimensional representation of motion and will be discussed further in the following section.

The general algorithms and abstractions for calculation of motion and the depth map have already been discussed in great detail in chapter one. Here, we will discuss the actual code used in determining the motion.

Flow Field Generation

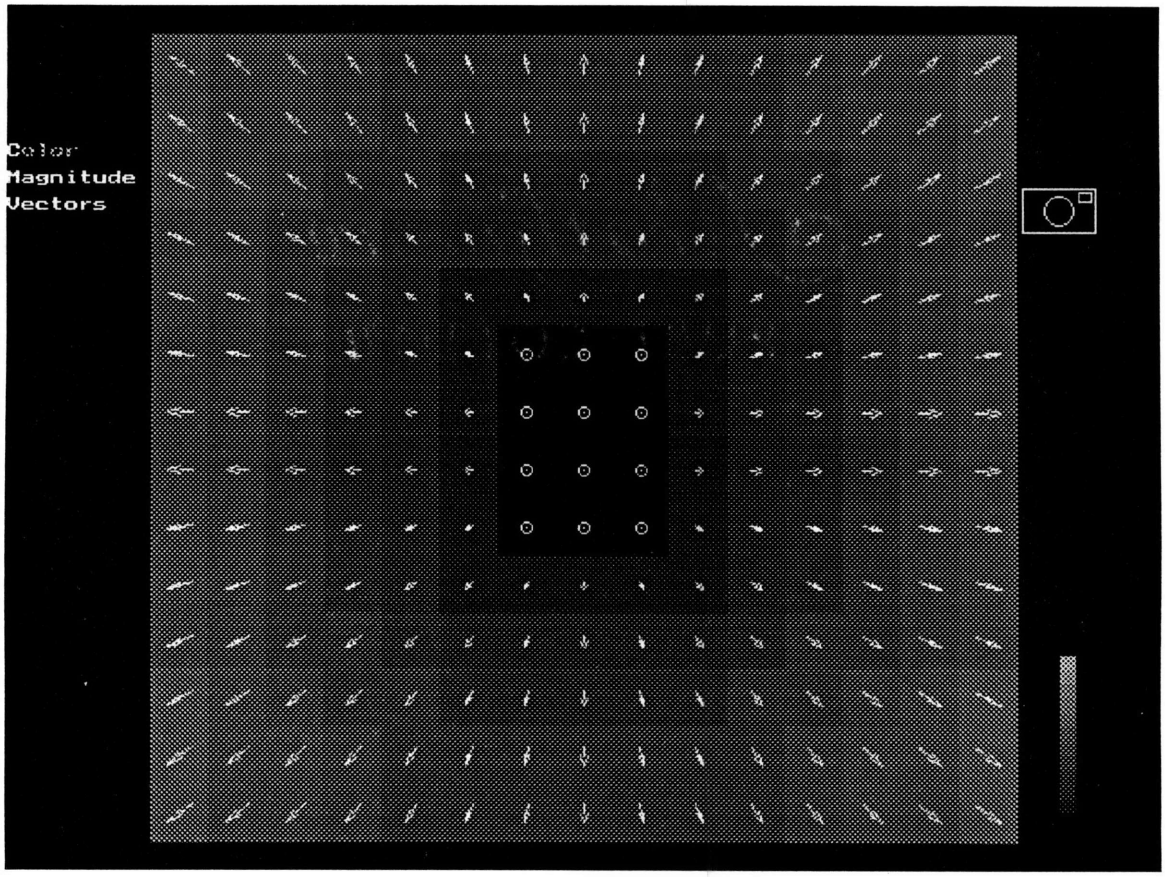


Figure 4.2 A Typical Flow Field Showing Motion Straight Forward

The flow field model is very simple. A typical flow field is show in figure 4.2. Each arrow, called a flow vector (see figure 1.7), in a typical flow field represents the displacement in that part of the frame due to a motion; every flow vector is generated

using the pattern matching algorithm in chapter one. The motion that created figure 4.2 is straight ahead -as if you were moving directly into the page. Straight forward motion is measured along the Z world coordinate axis or optical axis in figure 1.1. Another example of an actual flow field is the center image in figure 4.3. It is taken for granted that all flow fields, regardless of how they are generated, describe a certain motion for a specific duration of time. The flow fields in both Figure 4.2 and Figure 4.3 were both displayed using a graphical interface developed by Ely Wilson. The different levels of shading in both Figure 4.2 and Figure 4.3 represent the relative depth of the different portions of the screen; the brighter a block is the closer it is to the camera.

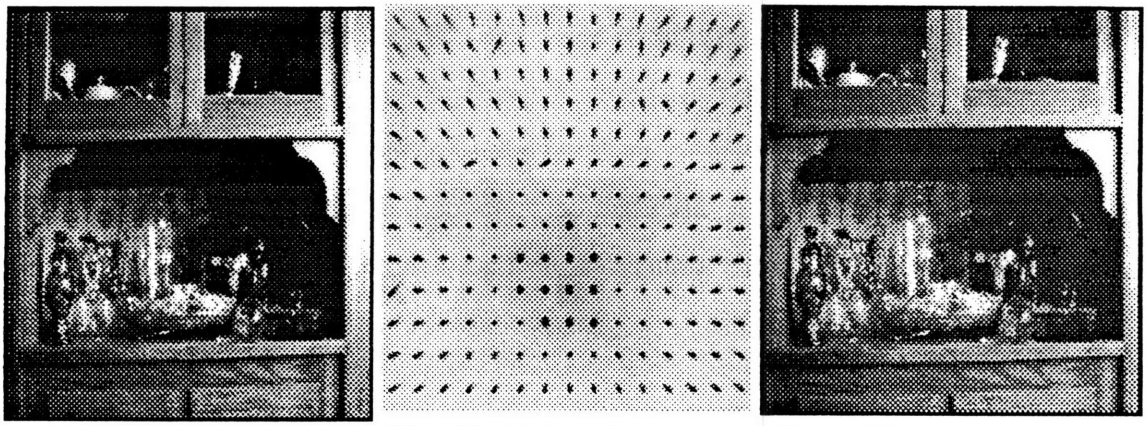


Figure 4.3 Actual Flow Field and Frames

A flow field is created by comparing two camera frames using the pattern matching algorithm of equation 1.15. In figure 4.3 the flow field, like in figure 4.2 was generated by straight ahead motion. The two frames used to create the flow field, the first and second frames, are only slightly different; the second frame is closer by three inches than the first frame. Hence the flow field indicates 3 inches of forward motion.

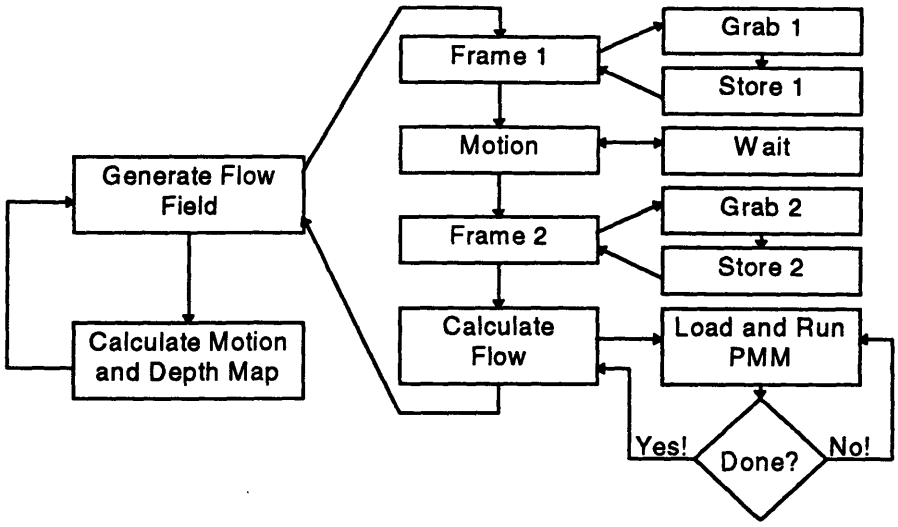


Figure 4.4 Simplified Subprocesses of Flow Field Generation

Figure 4.4 shows two simple expansions of our original flow generation step in figure 4.1. The first expansion delineates the natural boundaries for calculating the flow field that are evident from the discussion of figure 4.3. The natural expansion, however, is still too abstract to directly connect it to our prototype vision system. The second expansion begins to relate the flow generation process to the Vision System resources. Examining figure 4.4, the first resource that is used generating the flow field is the Frame Grabber.

4.2.1 Frame Grabber Control

The Frame Grabber, as was discussed in chapter 3, has a complete C library which makes control of it very simple. Once the Frame Grabber is addressed and plugged onto the computer bus the library can be used. All of the C functions utilized are either directly from the C library or made from functions within it.

The C function used to initialize the Frame Grabber is,

```
init_grabber();
```

Init_grabber() searches the bus for the Fame Grabber, enables it, prepares the necessary structures, sets the image resolution to be used and the location in the Frame Grabber memory where it will store pixel data.

The C function used to capture an image and hold it in the Frame Grabber memory is,

grab();

Grab() [11] digitizes and stores the black and white image being displayed on the Video Monitor, see figure 3.2, into memory; this is the "grab" step indicated in the third expansion of figure 4.4. The number of 8 bit pixels and the memory location where they will be stored was specified by init_grabber().

After grab() the frame data still resides in the Frame grabber memory. We need two functions to move the frames from the Frame Grabber into the computer memory; these are the two steps labeled "store" in figure 4.4. After the first frame is moved into the computer memory, see figure 4.3, it is split into data blocks and is sometimes referred to as the data frame. Likewise, when the second frame is moved into the computer it is broken into overlapping search windows and is often referred to as the search frame. The C function that we use to remove and store the data frame is,

store_data_frame(unsigned char** search_frame);

Store_data_frame(unsigned char** data_frame) removes the first frame from the Frame Grabber and stores it into a special structure called the data_frame.

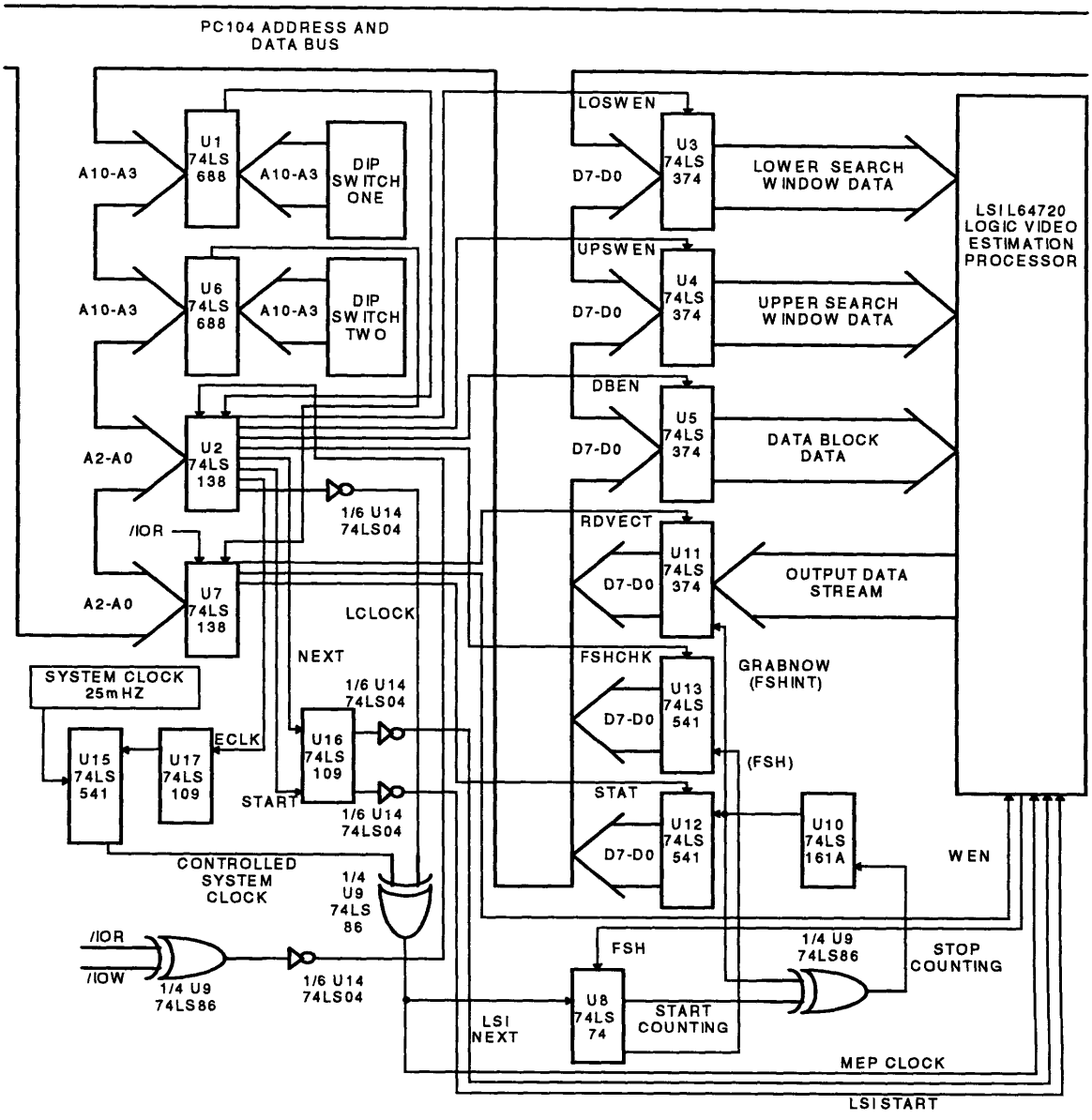


Figure 4.5 Functional Block Diagram of PMM

The C function that we use to remove and store the search frame is,

store_search_frame(unsigned char** data_frame);

Store_search_frame(unsigned char** search_frame) removes the second frame from the Frame Grabber and stores it into a special structure called the search_frame.

Complete control of the Frame Grabber is gained through these four functions. A detailed listing of each is provided in Appendix B.

4.2.2 Pattern Matching Module Control

The Pattern Matching Module (PMM), figure 4.5, is initialized and used in two function calls. The first function call is only needed once to initialize the PMM hardware after power up to put the hardware in a known state. The second function, which is called repeatedly, parses the data_frame and search_frame and produces the flow field which is stored into another special structure hardware_vector_array. The hardware_vector_array is simply an array of 210 characters where each character represents a flow vector. Each flow vector is a byte, eight bits, where the upper four bits are the x component, u, and the lower four bits are the y component, v; this allows each flow vector component to be positive or negative seven pixels. The function that initializes the PMM is,

```
initialize_LSI();
```

Initialize_LSI puts the PMM in a known state by resetting it's control and output logic, and latching in the hardwired Motion Estimation Processor (MEP) control registers.

The function that produces the hardware_vector_array is,

```
generate_flow_field( unsigned char** search_frame,
                     unsigned char** data_frame,
                     unsigned char*
                     hardware_vector_array);
```

Generate_flow_field(above parameters) loads the PMM with both a search window and a data block , waits for the PMM to compute the flow vector, stores the vector into hardware_vector_array, checks to see if it has collected 210 vectors, and then loads the PMM again or returns the flow field; this process is indicated in the last step of the third expansion of figure 4.4.

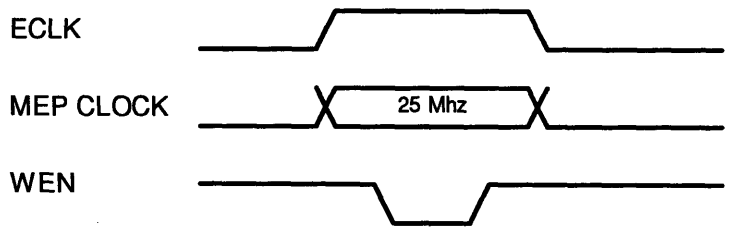


Figure 4.6 Relative Waveforms for MEP Initialization

The initialize_LSI() function is designed to reset the control and output logic and latch the control register of LSI L64720 MEP. The control latch configures the MEP for

application specific variables such as data block and search window size. As explained in section 3.6.2 most control functions have their own address. In order to assert an addressed control function all we have to do is merely access the appropriate address using outportb(correct address, don't care) or inportb(correct address); if we use outportb(correct address, don't care) it does not matter what we send as data since only decoding the correct address causes the desired action. The hardware, figure 4.5, is designed to automatically reset when a flow vector is read from it, therefore, all initialize_LSI() has to do to reset the control and output hardware is inportb(RDVECT) (note that RDVECT is REST1 in the schematic of Appendix A and in the code of Appendix B) and discard the inportb(RDVECT) result. Most control functions are asserted in the same way as RDVECT. To initialize the MEP we assert ECLK, which connects the 25Mhz clock to the MEP, assert WEN, which tells the MEP to latch the hardwired control registers on the next clock edge, and then turn off the system clock by unasserting ECLK; the relative waveforms for this operation are given in figure 4.6. The PMM is now ready for use by generate_flow_field(params).

The generate_flow_field(params) function is one of the largest functions used in the Vision System. Essentially, the generate_flow_field(params) function is the reason for the conception and design of the PMM hardware in figure 4.5. In the rest of this section we will discuss how this function interfaces and controls the PMM hardware of figure 4.5 after initialization. Please note that figure 4.5 is the same as figure 3.8; it is repeated here for convenience.

It has already been mentioned that the PMM hardware internally stores a data block and search window that it processes to generate a single flow vector. The PMM process is repeated two hundred and ten times to generate the flow field; this is the Calculate Flow block of Figure 4.4. Since calculating the flow field is a key process it would be useful for understanding the thesis work to describe how the PMM is loaded and run for a single flow vector computation within the function generate_flow_field(params).

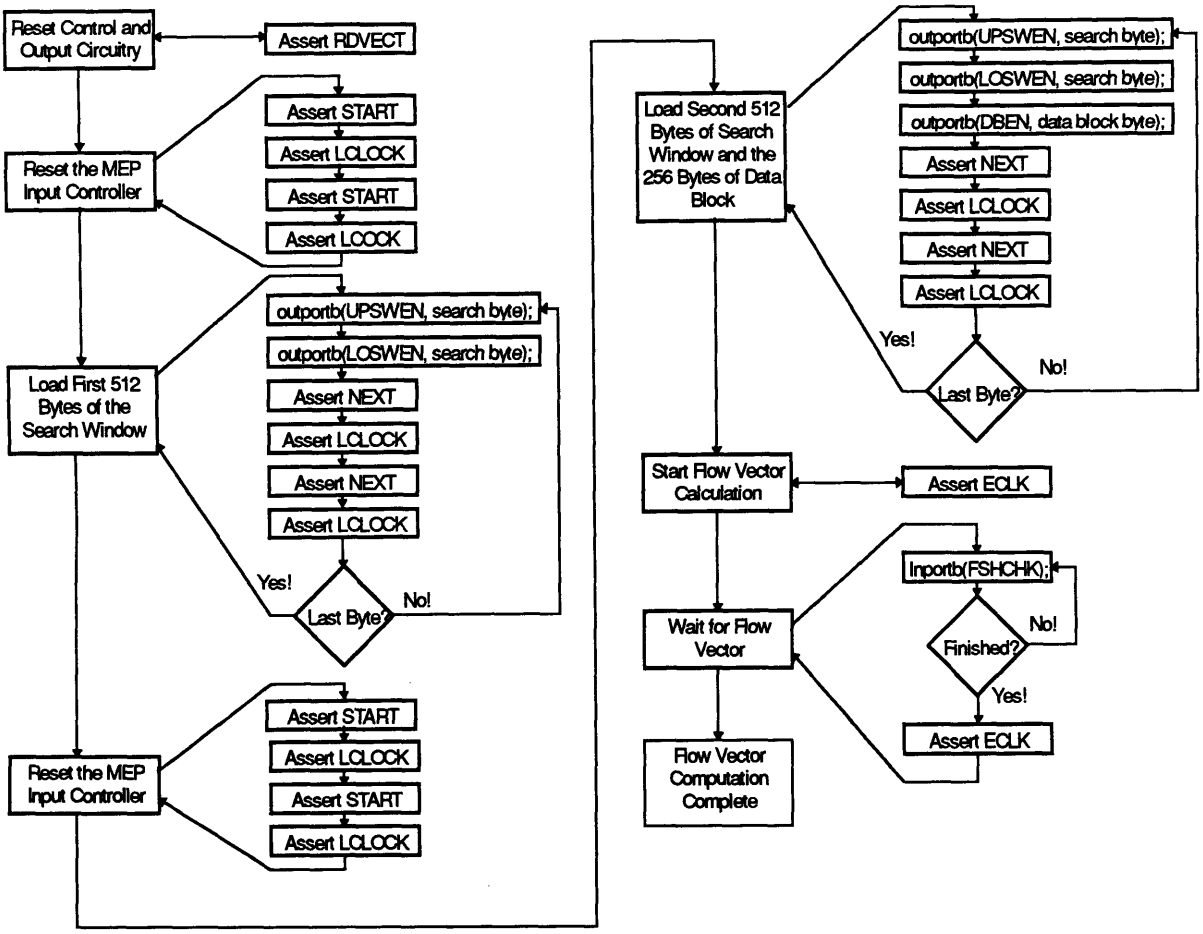


Figure 4.7 Single Flow Vector Computation

Generate_flow_field(params), besides parsing the data and search frames, makes repetitive calls to the functions full_compare_db_to_sw(params) and half_compare_db_to_sw(params). Full_compare_db_to_sw(params) and half_compare_db_to_sw(params) handle all of the interfacing with the PMM during flow field generation using the sequence of operations outlined in figure 4.7. The interface is quite simple using the PMM hardware which was intentionally designed to rely upon external software. The process for creating a single flow vector is depicted in figure 4.7; this detailed sequence could easily be performed by an embedded Finite State Machine (FSM) microcontroller or PROM as discussed in chapter 3 instead of using software.

The flow vector computation starts by asserting RDVECT which resets the control and output circuitry; this was discussed in the initialize_LSI() function. The introductory

reset is merely a precaution that guarantees the correct operation of the hardware even though it is automatically reset when a flow vector is read at the end of every cycle.

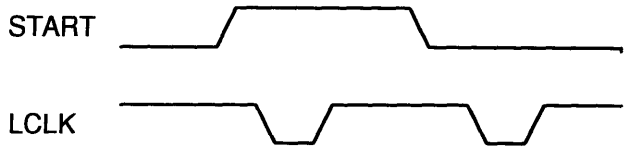


Figure 4.8 Relative Waveforms for Resetting the MEP Input Controller

Once we are sure that the logic is in the correct state we reset the input controller of the MEP so that we can start sending it pixels. The way the input controller is reset is similar to the way the MEP was initialized in figure 4.6. The input controller is reset by asserting START clocking the MEP using LCLOCK and then unasserting START and clocking the MEP again with LCLOCK; this is show in figure 4.8.

The third and fifth operations load the MEP. The third operation loads the MEP with first five hundred and twelve bytes of the search window and the fifth operation loads the MEP with the second five hundred and twelve bytes of the search window with 256 bytes of the data block. Loading the search window is divided into two steps because the search window is 4 times as large as the data block. In both loading steps two bytes of the search window are loaded at a time where only a single byte of the data block is loaded at a time during the second load step. (See Figure 4.5) Consequently, for each load operation there is 256 iterations. In each iteration of a load the necessary bytes are latched into the corresponding 74LS374's and then clocked into the MEP all at once by asserting NEXT and LCLOCK. The NEXT-LCLOCK waveforms are exactly the same as in figure 4.8 with NEXT interposed for START. The input controller is reset between the two loads because the input logic can only go through 256 iterations before it has to be restarted, see figure 4.7

After a complete search window and data block have been loaded into the MEP, ECLK is asserted which connects the 25Mhz clock to the MEP; this allows the MEP to compute the flow vector much more quickly than a single microprocessor could.

The last step in the flow vector computation, figure 4.7, is to wait for the result. The PMM sets the lowest bit of U13 high when it is finished computing and the flow vector is ready to be read using inportb(RDVECT). Therefore, the system continuously

checks to see if the MEP has finished by reading U13 with inportb(FSHCHK). Once the MEP is finished the system reads the flow vector with inportb(RDVECT).

It is important to understand the creation of a flow vector in order to understand how the PMM hardware is used in the generation of the flow field.

4.2.3 Flow Field Calculation

We have all the pieces of software to fill in figure 4.4. The sequence of operations to generate the flow field is,

```
grab();
store_data_frame(data_frame);
    (Any Loop Structure to Form a Delay for Motion to
    Occur.)
grab();
store_search_frame(search_frame);
generate_flow_field(search_frame,
                    data_frame,
                    hardware_vector_array);
```

Thus our abstractions for flow field calculation are complete! An abstraction, however, is only as good as ability to assist understanding and logical thought; if an abstraction is too ambitious it will render itself useless by hiding to much detail. The sequence above contains only a few functions each with a distinct character and clear purpose. The individual instructions each provide a complex function and are easy to use; they only hide the details about data parsing and loop control. Furthermore, every function is already a combination of recursive and/or iterative sub-function calls. Thus, in this situation, combining the flow field sequence into a single function would not yield a better abstraction, but a less useful one.

Motion Calculation

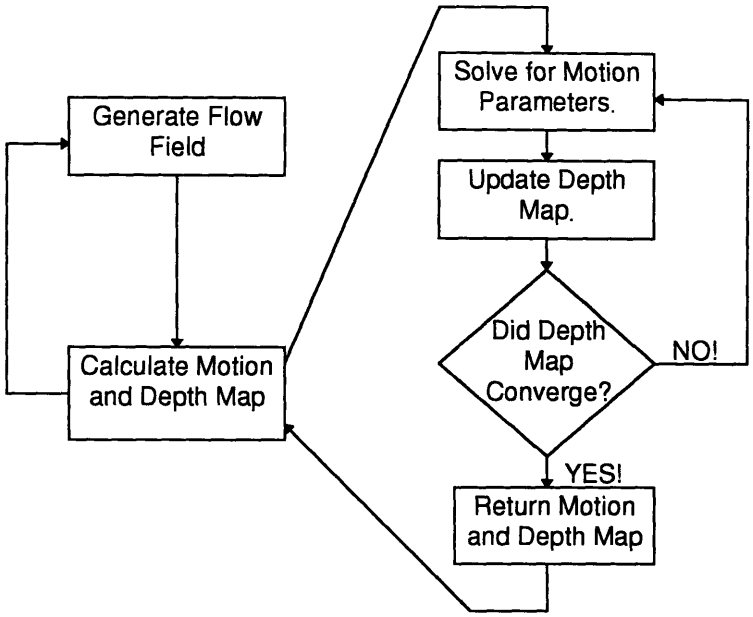


Figure 4.9 Simplified Subprocesses of Motion Calculation

The equations for calculating motion are thoroughly discussed in chapter one. Calculating motion, unlike the generation of the flow field, does not require a piece of hardware external to the main computer; it is performed completely in software.

The bulk of the computations to calculate motion are matrix manipulations of doubles. The matrix library that is used in the code was found on the Internet; it was developed by Patrick Ko Shu Pui for educational use only. The format of the matrix library source code was changed so that it could be more easily incorporated into Borland C++ 4.51, it is not included in the thesis.

4.3.1 Motion Calculation Functions

The outline of the major components in the motion calculation process are shown in figure 4.9, which is a combination of figure 1.6 and 4.1. Since the development is entirely in software we can directly implement every step in the expansion of the motion calculation of figure 4.9. The implementation is straightforward and we start with solving for the motion parameters.

$$m = \left(A^T A\right)^{-1} A^T b$$

Equation 4.1 Least Squares Motion Estimation Equation

Equation 1.24 is repeated here as equation 4.1 for convenience. Initially, we can only solve for the motion parameters if we make assumptions about the values in the depth map; this is because we directly use the depth map to create the matrix A, in equation 4.1 which is defined in equation 1.19. A conservative depth map assumption is to set all of the depth values equal to one. Once we have assumed something for our depth map we create A and b, which is the flow field, and solve for the initial motion; m as defined in equation 1.17 and b as defined in equation 1.20.

The data structure that is used in the matrix library is called a MATRIX. After we have created our matrices, we must initialize them before calculating for the motion. The C functions that we use to fill A, b, and our initial depth map are,

```
fill_b_and_depth_matrix(MATRIX b,
                        MATRIX depth_map,
                        unsigned char* hardware_vector_array);
fill_A_matrix(MATRIX A, MATRIX depth_map);
```

Fill_b_and_depth_matrix(params) takes the flow field, which is stored as eight bit vectors in hardware_vector_array, separates and stores their u and v components in b and initializes all of the depth values in the depth map.

Fill_A_matrix(MATRIX A, MATRIX depth_map) calculates the elements of A as defined in equation 1.19; the values of x and y in the element calculation are the actual locations of the pixels created by the frame grabber on the CCD normalized by the focal length of the camera lens. The pixel locations are discussed in the calibration section of chapter five.

After we have our A, b, and the depth map it is a simple matter to compute motion. Using our matrix library, equation 4.1 is represented as,

```
m = mat_mul(mat_inv(mat_tran(A), A)),mat_mul(mat_tran(A), b));
```

The individual functions are,

```
mat_mul(MATRIX x, MATRIX y);
mat_inv(MATRIX x);
mat_tran(MATRIX x);
```

The mathematical operation of each of these functions is clear, mat_mul(params) multiplies two matrices, mat_inv(params) inverts a matrix, and mat_tran(params)

transposes a matrix. Altogether these functions make the motion computation a single line of code.

As indicated in figure 4.9, after every motion iteration we update the depth map from the motion parameters using,

$$Z = \frac{(-U + xW)^2 + (-V + yW)^2}{\left(u - \left(xyA - \left(x^2 + 1\right)B + yC\right)\right)(-U + xW) + \left(v - \left(\left(y^2 + 1\right)A - xyB - xC\right)\right)(-V + yW)}$$

Equation 4.2 Update Depth Map Equation

Equation 1.25 is repeated here as equation 4.2 for convenience. Again, the implementation is very direct, we use equation 4.2 to recalculate each depth value using the new motion parameters. Equation 4.2 is implemented in C almost exactly as it appears above. All of the capitalized variables in equation 4.2 are components of motion and every lower case variable is location on the CCD image plane normalized by the camera focal length. The depth map is updated by an iterative loop that computes equation 4.2 for each flow vector in the flow field; in our case there are 210 depth values.

As the depth map is updated, the largest absolute error between the new and old depth values is recorded; this is our depth convergence value and is termed depth_convergence in the code. The depth convergence is compared with the convergence tolerance, denoted convergence_tolerance, to determine whether another set of motion parameters should be calculated. If the depth_convergence value of the current depth map is less than or equal to the convergence_tolerance value then it is implied that the current depth map has converged. Once convergence has been achieved it is implied that the motion parameters derived from the depth map and the values in the depth map accurate enough to be returned for postprocessing by the user interface.

The entire motion calculation process shown in figure 4.9 is compacted into the single function,

```
MATRIX calculate_motion(unsigned char* hardware_vector_array,
                        double* depth_map,
                        double convergence_tolerance);
```

Calculate_motion(params) performs all the necessary memory allocations and loops to compute the motion parameters and depth map. It returns the motion parameters in a $6x1$ matrix of the form in equation 1.17 and updates the depth map pointed to by the array of doubles, depth_map.

Chapter Summary

This chapter discussed all of the underlying processes developed and implemented in the prototype machine vision system. A simple two step abstraction of all of the software is given in figure 4.1. The first task for the software system is to generate a flow field using the Pattern Matching Module described in figure 4.5. The second task for the software is to iteratively compute the motion parameters and depth map. The combination of these two steps is implemented in the following C abstractions:

```
grab();
store_data_frame(data_frame);
    (Any Loop Structure to Form a Delay for Motion to
    Occur.)
grab();
store_search_frame(search_frame);
generate_flow_field(search_frame,
                    data_frame,
                    hardware_vector_array);
calculate_motion(hardware_vector_array,
                    depth_map,
                    convergence_tolerance);
```

Where each of the functions perform the following:

grab() captures a video image.

store_data_frame(params) and **store_search_fram(params)** take the currently captured video image and move it into a usable data structure.

generate_flow_field creates an estimate of the motion field, called a flow field, from two consecutive video images.

calculate_motion(params) determines three dimensional translation and rotation from a flow field.

Each function is fundamentally and functionally distinct. Together, all of these functions are an implementation of the processes used for calculation of depth and motion in the prototype vision system.

CHAPTER FIVE
CALIBRATION, RESULTS, AND PERFORMANCE

This chapter discusses the major facets of the Vision System calibration, performance, and results. Naturally, the first section describes how and why the Vision System is calibrated. The second section provides our first taste of some real results followed by an analysis of those results. As a consequence of the initial test results, the chapter takes a momentary detour to discuss the average vector elimination flow field filter. After the average vector elimination filter is included in the motion parameter computation the final results are presented. The last section of the chapter tabulates the speedup of the different components of the Vision System over the duration of this thesis and discusses how the Vision System might be improved.

5.1 System Calibration

We have the system, the theory, and an understanding of the limitations of both. Now it is time to make the system reflect a reality; this is accomplished through calibration. Our understanding provides the lens through which we will calibrate our equations. A critical test of understanding for a complex system is a clear identification of units.

5.1.1 Scale Factors

All of the equations discussed in chapter one provide us with six motion parameters. The six motion parameters are U, V, W, A, B, C. The translations, U, V, W are in the units of the principle distance and defined along the X, Y, Z coordinates. In our case the units of the principle distance will be meters. The rotations, A, B, C are the right hand rotations around X, Y, Z in radians respectively. The equations actually give us rotations in radians, but we need external information to handle the translations. The iterative calculation of the six parameters has a whole plane of solutions differing by a relative scale factor; the scale factor can be applied to either the depth map and hence the

motion parameters or the motion parameters directly. The need for a scale factor can easily be seen when one imagines that if the external world were twice as big, and our movement through it were twice as fast, we would generate the same motion field; the relationship between the size of the external world and our velocity is our plane of solutions. The Vision System itself needs knowledge of either a depth to a point within its depth map or an actual value of a motion parameter in order to make the calculated translations useful. The known depth value would allow us to correctly rescale the whole depth map and then generate real translation parameters. A known real world translational value would be directly compared to its generated counterpart to find a scale factor for the other two translational components. The translational scale factor should be constantly recalculated and applied for every motion parameter generation; this is external updating calibration. In a real application it was proposed that an optical encoder be used to measure displacement from which the translation along one of the three axes could be inferred; in all of the calculations here the details of the depth map and its scale factor are suppressed in favor to the simpler method of measuring an actual displacement along an axis. The rotations, however, get their values from parameters that are embedded in the actual equations; these are permanent calibrations assuming that we know the principle distance and the pixel spacing.

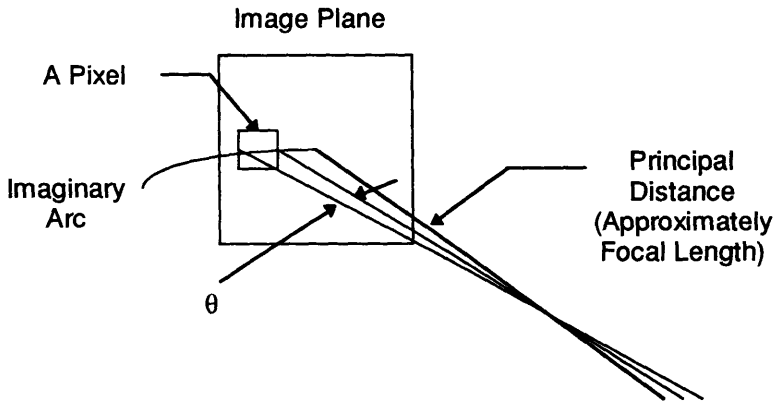


Figure 5.1 Angular Resolution

$$s = r\theta \;\Leftrightarrow\; \theta = \frac{s}{r}$$

$$angular\ resolution = \frac{\approx pixel\ width}{principal\ distance} = \frac{\approx pixel\ width}{\approx focal\ length}$$

Equation 5.1 Angular Resolution

The rotations are based upon the apparent horizontal and vertical width of an image pixel (See Figure 5.1). The horizontal and vertical angular resolutions of the system are the horizontal and vertical pixel width divided by the principal distance which is approximately camera lens focal length (See Equation 5.1); this assumes that the horizontal and vertical pixel width is sufficiently small compared to the focal length. The beautiful part of this is that as long as we determine the pixel size correctly, rotations should thereafter remain calibrated; the pixel size, however is not an easy quantity to measure and will be referred to as a "virtual" pixel to imply its mysteriousness.

5.1.2 Determining The Virtual Pixels

We can call the parameter we need to measure a "virtual" pixel because it is the result of a series of different conversions. It is easiest to explain the virtual pixel by following the flow of the image data. First, a CCD is charged by impinging light which is serially shifted out, low pass filtered, and forced into an ugly analog video standard at the camera's internal clock rate. Second, a frame grabber samples the analog video signal at a its internal clock rate and drops it into memory. A key conceptual point is that the definition of a virtual pixel is heavily dependent upon the hardware used to generate it. In our system the camera has a moderate resolution of 768 pixels horizontally by 494 pixels vertically. The frame grabber, however, generates its own number, which is less than the actual image resolution, of both horizontal and vertical pixels from the video signal. Further, in able to process the images faster we use the frame grabber on low resolution which further reduces both the number of horizontal and vertical pixels that make up a grabbed image. Thus the real size of the virtual pixels is a complicated issue. It is obvious that we have a choice between spending an indeterminate amount of time measuring lots

of signals or examining the flow field output and relating it to the objects within its view. The second option was used because the empirical result is all the we need. If we were attempting to create or modify the interface between the camera and frame grabber we would need to verify and catalog all of the signals.

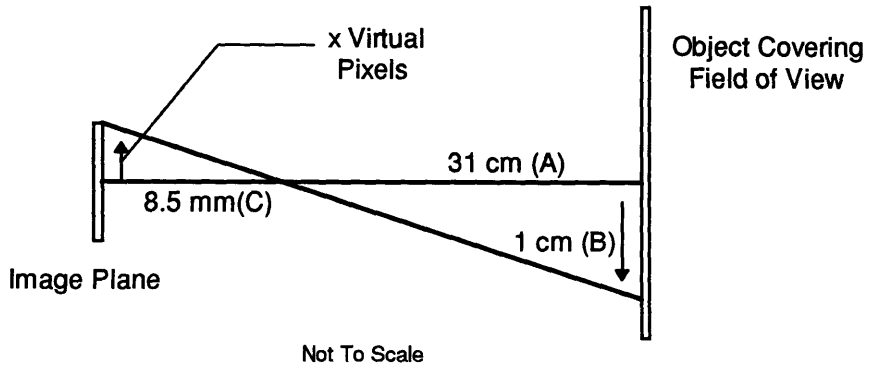


Figure 5.2 Pixel Rate Calibration

$$\frac{scale[meters/pixel]*x}{C} = \frac{B}{A} \Leftrightarrow scale[meters/pixel] = \frac{BC}{Ax}$$

Equation 5.2 Pixel Scale Factor

Relating the flow field to the objects within its view was done in two ways. The first way is using the rate of change of an object captured by the flow field. The way this works is shown in Figure 5.2. We encompass the entire field of view of the camera with a high contrast object at a known distance, A, from the camera or surface. Once the object is in place and the distance has been measured an image is grabbed, the object is moved a known distance, B, and a second image is grabbed. After the flow field is generated we can calculate Equation 5.2 to determine the number of meters per pixel. Figure 5.2 describes the actual motion and distances use to calculate the horizontal and vertical pixel widths. A note should be made that the lens that was used has a very small thickness parameter such that the simple diagram in Figure 5.2 suffices; the thin lens approximation, however, is not always true -the input node and output node of a lens do not have to coincide.

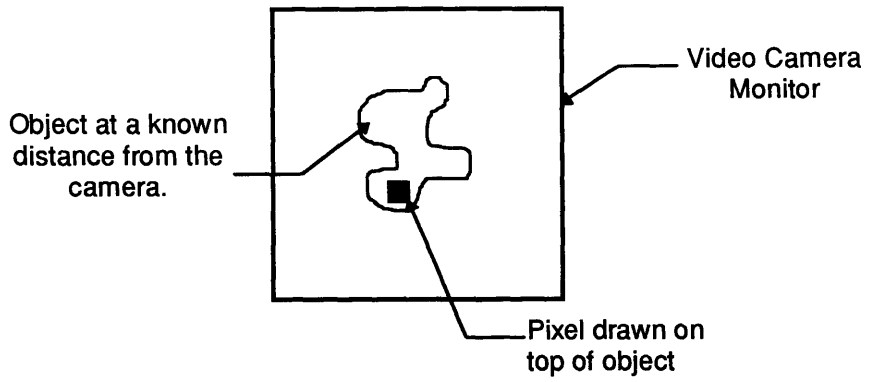


Figure 5.3 Pixel Fill Calibration

The second way to calibrate the horizontal and vertical width of a virtual pixel is essentially the same as the pixel rate calibration and we will use the same variables as shown in Figure 5.2. First, position an object at a known distance, B, in Figure 5.2. Second, physically measure some part of an object, C, in the camera's view, and then overwrite the camera image pixel by pixel until the measured area is filled in, x, (See Figure 5.3). Now you can use Equation 5.2 to calculate for the scale factor.

Unsurprisingly, both methods yielded the exact same results. The horizontal and vertical scale factors are,

$$\textit{Horizontal Scale Factor} \quad = 62.5\ \mu m\ per\ pixel$$

$$\textit{Vertical Scale Factor} \quad = 50.0\ \mu m\ per\ pixel$$

The surprising part of this measurement is that we would expect the vertical scale factor is so large; this value is approximately 5 times the height of a real pixel in the camera!

The horizontal and vertical pixel width are the only calibration coefficients other than the external translation value that we need to correct and calculate our six motion parameters and the associated depth map.

5.2 Initial Results

Each trial in the initial results is based upon sets of data from three sources. The first source is the actual controlled motion that has been incurred for the trial. The second

source of data is the results that we get from moving the camera through the "motion" given by the motion parameter entries and letting the prototype try to calculate the correct results. The last source of data is the simulation data. The only difference between the simulation and the prototype is where the flow field information comes from. In the prototype the flow field is generated by dedicated hardware, in the simulation the flow field is computed using the Longuet Higgins, and Pradzny equations (Equations 1.14). All of the trials have been with the camera examining a flat surface with a high contrast about one meter away from the lens. All of the translation entries were scaled using an empirically derived scale factor.

INITIAL MOTION PARAMETER RESULTS

Trial #	System	U[cm]	V[cm]	W[cm]	A[rad]	B[rad]	C[rad]
1	motion	0	0	0	0	-0.026	0
	prototype	0.163	0.343	-0.971	0.009	0.021	0.005
	simulation	0	0	0	0	-0.026	0
2	motion	0	0	0	0	0.026	0
	prototype	1.23	0.2	0.257	-0.005	-0.018	-0.007
	simulation	0	0	0	0	0.026	0
3	motion	0	0	0	0	0	0
	prototype	0.029	0.057	0	0	0	0
	simulation	0	0	0	0	0	0
4	motion	2	0	0	0	0	0
	prototype	-0.086	1.267	-0.029	-0.013	-0.028	-0.005
	simulation	1.911	0	0.229	0	0.017	0
5	motion	0	0	2	0	0	0
	prototype	0.286	0.886	-0.857	-0.009	0.002	-0.002
	simulation	0	0.171	1.911	0	0	0
6	motion	0	2	0	0	0	0
	prototype	-0.343	1.514	0.571	-0.008	0.009	-0.016
	simulation	0	1.911	0.229	0	0	0

Table 5.1 Initial Motion Parameter Calculations

Table 5.1 displays the Vision System data for six different test trials. It should be noted that the simulation was an essential development in order to debug the motion calculation code.

5.2.1 Initial Results Analysis

Although it is refreshing to examine actual test data, after a cursory glance at the data in Table 5.1 it is obvious that the prototype is not operating properly while the simulation is doing well. The prototype translations are almost always wrong and appear to be in the opposite directions. The rotations are much more sensitive than the translations but we can again observe that the magnitudes of the results are in the wrong direction. A portion of the prototype error is, of course, because of positional errors during testing but this does not explain why the directions are inverted. The simulation, however, gives good repeatable estimates of the motion parameters.

As explained earlier the only difference between the simulation and the actual prototype is where we get the flow field. In the prototype the flow field is created by physical camera motion. In the simulation the flow field is created using the Longuet, Higgins, and Pradzny equations to a fair degree of resolution.

The prototype problem is twofold. First, it is a resolution problem. The prototype generates flow fields with 256 different flow vectors where the simulation can create flow vectors to a few decimal places. The lack of quality in the flow field is due to the integer measuring ability of the matching algorithm [12]. Although the issue of flow vector resolution and pixel interpolation is applicable here it was determined through extensive testing that poor flow field quality was the major source of coarse error and resolution was left as a system refinement.

Poor flow field quality is when a flow vector is either pointed incorrectly or has the wrong magnitude. Figure 4.3 shows a flow field that has flow field errors. Every flow field generated by the Pattern Matching Module typically has a few incorrect flow vectors. The incorrect flow vectors are dependent on the environment and the pattern matching algorithm. The only way to reduce the incorrect flow vectors is to perform flow vector filtering or change the hardware. Flow vector filtering compares each flow vector to its neighbors in order to determine if it is incorrect. One flow vector filtering technique is average vector elimination which is discussed next.

5.3 Average Vector Elimination Filter

Most flow fields that are generated have erroneous flow vectors which result in incorrect computations for the motion parameters (See Figure 5.4). The goal of applying a filter to the flow field before the motion parameters are calculated is to eliminate flow vectors that are not representative of the overall flow and would affect the computation.

The idea of using the average vector elimination filter is to increase the accuracy of the motion parameter computation.

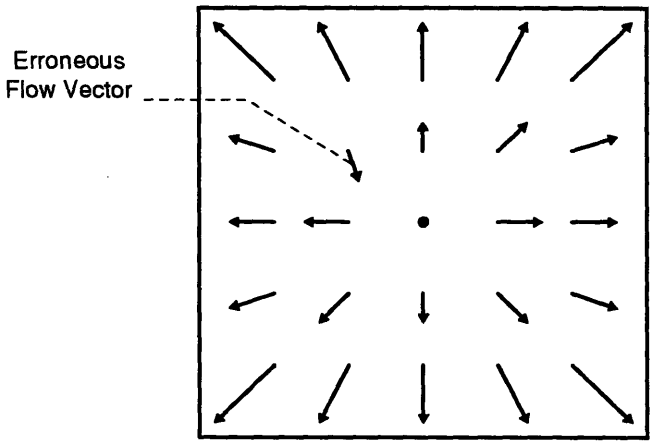


Figure 5.4 Flow Field with an Error

The filter that was implemented in this thesis is a average vector elimination filter. Each flow vector has a vertical and horizontal component, u & v respectively as defined in Figure 1.7 and Equation 1.16. The components of each flow vector are compared to the component average of its neighboring flow vectors. Typically there is usually four neighboring flow vectors above, below, left, and right of the flow vector under examination; however, the flow vectors around the perimeter of the motion field are compared to as many neighbors as there are available. If the absolute difference between a flow vector's component and the average for that component is not within a tolerance, called pixel tolerance since the flow vector components are measured in terms of pixels, the flow vectors is neutralized, that is set to zero. Although the nulling of a flow vector adds another constraint to the least squares computation, for the purposes of this thesis it

is a suitable approximation; it is comforting to know that theoretically we only need 7 flow vectors to generate a unique solution of the motion equations. The pseudocode for the local averaging filter is simple and follows,

$$\bar{u} = \frac{\left(u_{i,j+1} + u_{i+1,j} + u_{i-1,j} + u_{i,j+1}\right)}{4}$$

$$\bar{v} = \frac{\left(v_{i,j+1} + v_{i+1,j} + v_{i-1,j} + v_{i,j+1}\right)}{4}$$

$$\left|u_{diff}\right| = \left|\bar{u} - u\right|$$

$$\left|v_{diff}\right| = \left|\bar{v} - v\right|$$

$$if\left(\left(\left|u_{diff}\right| > pixel\ tolerance\right) or \left(\left|v_{diff}\right| > pixel\ tolerance\right)\right)$$

$$u = 0 \quad \& \quad v = 0$$

The average vector elimination algorithm is implemented in the single C function,

```
void local_averaging_filter(char *vect_comps, double
                                        pixel_tolerance);
```

5.4 Final Results

The meaning of the entries in the following table are the same as in Table 5.1.

FINAL MOTION PARAMETER RESULTS

Trial #	System	U[cm]	V[cm]	W[cm]	A[rad]	B[rad]	C[rad]
1	motion	0	0	0	0	-0.026	0
	prototype	0.943	0..029	0.086	0.001	-0.024	-0.006
	simulation	0	0	0	0	-0.026	0
2	motion	0	0	0	0	0.026	0
	prototype	0.943	-0.2	0.243	-0.001	0.024	0.01
	simulation	0	0	0	0	0.026	0
3	motion	0	0	0	0	0	0
	prototype	0	0	0	0	0	0
	simulation	0	0	0	0	0	0
4	motion	2	0	0	0	0	0
	prototype	2.01	0.171	-0.086	-0.003	-0.013	0.002

	simulation	1.911	0	0.229	0	0.017	0
5	motion	0	0	2	0	0	0
	prototype	0.029	0.251	1.657	-0.007	0	0.001
	simulation	0	0.171	1.911	0	0	0
6	motion	0	2	0	0	0	0
	prototype	0.086	2	0.251	-0.007	0.003	0.005
	simulation	0	1.911	0.229	0	0	0

Table 5.2 Final Motion Parameter Results

5.4.1 Final Results Analysis

The meaning of the entries in Table 5.2 is the same as Table 5.1, but the data is very different. The local averaging filter, without any other changes, significantly improved the motion parameter results. The most obvious improvements are that now all of the parameter estimates are in the correct direction and all of the estimated parameters are more accurate.

5.5 System Speedup

Speedup is the amount the execution time of a system or operation has decreased. The measure of speedup applied to this thesis is Amdahl's law. Amdahl's law captures, in absolute terms, the most unambiguous meaning of speedup; the law is a ratio of execution times with and without enhancement,

$$Speedup = \frac{Execution\ time\ for\ entire\ task\ without\ enhancement}{Execution\ time\ for\ entire\ task\ with\ enhancement\ when\ possible}$$

Equation 5.3 Amdahl's Law [13]

Amdahl's law is the heart of our Speedup analysis.

SPEEDUP IN FLOW FIELD GENERATION

Flow Field Generation	386 25Mhz PC104 w/o Math	Flow Field Generation	Speedup

	Coprocessor	Hardware	
Average Execution Time	$\geq 312\ s$	$< 250\ ms$	$\frac{\geq 312\ s}{< 250\ ms} \Rightarrow > 1248$

Table 5.3 Flow Field Speedup

The largest speedup that we have achieved in the work described in this thesis is the improvement in the time it takes to create the flow field. Traditionally, this is usually the data flow bottleneck. The traditional approach is shown clearly in the 312 second flow field generation time for a 386 processor shown in Table 5.3. The problem is that the flow field cannot be efficiently calculated with a single sequential processor. At first glance it might have been thought that the inefficiency is that the 386 does not have a math coprocessor; this is an incorrect assumption because the flow field hardware is restricted to generating flow vectors with a magnitude of 256 or less! Our flow field computations are strictly integer operations and therefore get the full attention of the processor without the need for messy floating point interrupt handlers. The traditional bottleneck in the flow field generation is not the speed of an individual processor but that the shear bulk of information that should be processed by a parallel network of processors.

So our speedup as calculated in Table 5.3 is 1248 or greater; this is a reasonable speedup when it is considered that the hardware is running at, 25Mhz, 15 Mhz beneath it potential top speed. Of the 250 millisecond latency it has been estimated that 75% of the delay is due to the data bus transaction time. In the original concept the Pattern Matching Module would have a direct link to the frame grabbing system and its own fast memory for storing and accessing the image data independent of the data bus. In a self contained frame grabbing and processing system the flow field latency could be reduced to less than 60 milliseconds and with a faster clock approach real time flow field generation. Flow field resolution can be increased by merely adding another Motion Estimation Processor in the PMM or by performing pixel interpolation -the possibilities are endless! Since our Vision

System was able to create flow fields at an abundant rate the real bottleneck in this Vision System is in the motion parameter and depth map calculation.

SPEEDUP IN MOTION COMPUTATION

Motion Computation	386 25Mhz PC104 w/o Math Coprocessor	486 66Mhz PC with Math Coprocessor	Speedup
Average Execution Time (for a single iteration of convergence loop)	$\approx 10s$	$< 0.125\,s$	$\frac{10\,s}{< 0.125\,s} \Rightarrow > 80$

Table 5.4 Motion Computation Speedup

It was known from day one of this project that there was going to be a section dedicated to discussing the computation time of the motion parameters. The only practical approach to developing code for the motion parameter computation was to make every new parameter and every associated declaration a floating point number. The approach to the system was "create an operational system before you create an efficient one." As the code came together it was possible to eliminate a few of the floating point numbers -but the code efficiency issues were not deeply probed -that is an area of on going research. Our development platform is a 386 25Mhz PC104 processor without a math coprocessor; this means that for every floating point operation a minimum of 25 integer operations have to be executed. The saving grace is that all of the code and hardware is portable and, when we obtain a 200Mhz x86 PC104 with a coprocessor it can be plugged in to create a faster system. Table 5.4 lists the speedup for the motion parameter computation running on a moderate desktop computer where the speedup is 80 times; the desktop computer with the specifications listed in Table 5.4 was used because it is comparable to one of the slowest PC104 computers available with a floating point coprocessor. A key point to remember is

that the motion computation is now the bottleneck. Therefore, for all of our work, we have the potential speedup of at least a 80 times.

THE MINIMUM OVERALL EXECUTION TIME

Minimum Overall Execution Time	386 25Mhz PC104 w/o Flow Field Hardware and w/o Math Coprocessor	386 25Mhz PC104 with Flow Field Hardware but w/o Math Coprocessor	486 66Mhz PC with Flow Field Hardware and Math Coprocessor
Flow Field Generation	$\geq 312\ s$	$< 250\ ms$	$< 250\ ms$
Motion Parameter Computation (per iteration)	$\approx 10s$	$\approx 10s$	$< 125\ ms$
Total Average Execution Time (for flow field generation and first iteration of motion parameters.)	$\geq 322\ s$	$\approx 10s$	$< 375\ ms$

Table 5.5 Minimum Overall Time

Table 5.5 indicates that the minimum overall execution time with the Vision System running on the cheapest adequate processor is 375 milliseconds; this is rapidly approaching the real time application boundary!

5.6 Chapter Summary

The chapter opens discussing the units of the motion parameters where the translational components are in the units of meters and the rotational components are in radians. The need for an external reference to scale either the translational components or the depth map is explained and the two internal calibration parameters, the horizontal and

vertical virtual pixel width, are introduced. The two redundant ways that were used to determine the virtual pixel widths that are used as calibration parameters are each explained and the final virtual pixel width values are given:

$$\textit{Horizontal Scale Factor} = 62.5\ \mu m\ per\ pixel$$

$$\textit{Vertical Scale Factor} = 50.0\ \mu m\ per\ pixel$$

Following the calibration section is the first set of experimental data. The data indicated that the simulation was functioning well, but that the prototype was not. The prototype errors were that it gave incorrect directions, that is minus signs, and some of the parameters had inaccurate magnitudes. The fact that the simulation performed better clearly indicated that the motion parameter computation algorithm was sound and that the difference was in something the prototype lacked -flow field quality. Although the motion parameter algorithm will produce the correct answers given the correct flow field, it has a very low tolerance to flow field anomalies; this was not an intuitive result since the motion calculation algorithm is based upon a least squares solution. The prototype flow field quality was substantially improved by filtering, that is setting to zero, any flow vector that had an orientation or magnitude that was inconsistent with its neighbors. The nulling of the flow vectors, however, adds a new constraint to the least squares solution; a filter that replaces the incorrect flow vector with the correct average values may be a better solution [14]. The flow field filter that was used is a average vector elimination filter.

The average vector elimination filter calculates the averages of the vertical and horizontal flow vector components surrounding the vector under examination. If the components of the flow vector under study are different than the averages by a specified number of pixels, that is a pixel tolerance, the vector is nulled. The empirical pixel tolerance used in the local averaging filter was four.

After the average vector elimination filter was incorporated into the motion calculation algorithm, the motion parameter results improved dramatically. The improved the flow field quality enough so that not only are the parameter estimates in the correct

directions, but they are very nearly the correct magnitudes. Perhaps now it would be useful to increase the flow vector resolution.

The last section of the chapter is dedicated to the most tangible result of the prototype system which is its improvements in speedup. The prototype hardware creates flow fields 1248 times faster than when the thesis was started. Perhaps what is even more startling is that with small improvements in equipment the total Vision System latency to produce motion parameters from camera motion can be reduced to less than 375 milliseconds! It would appear that machine visions system will soon be a practical addition if not an alternative real time instrument.

CHAPTER SIX
CONCLUSION

The main focus of this thesis has been to engineer and prototype an operational Machine Vision System for Autonomous Applications to serve as a proof of concept system. Among the requirements placed upon the Vision System, the requirements that it be physically small, and produce data quickly have been the most challenging and influential. The work within the thesis is split between digital hardware and software design.

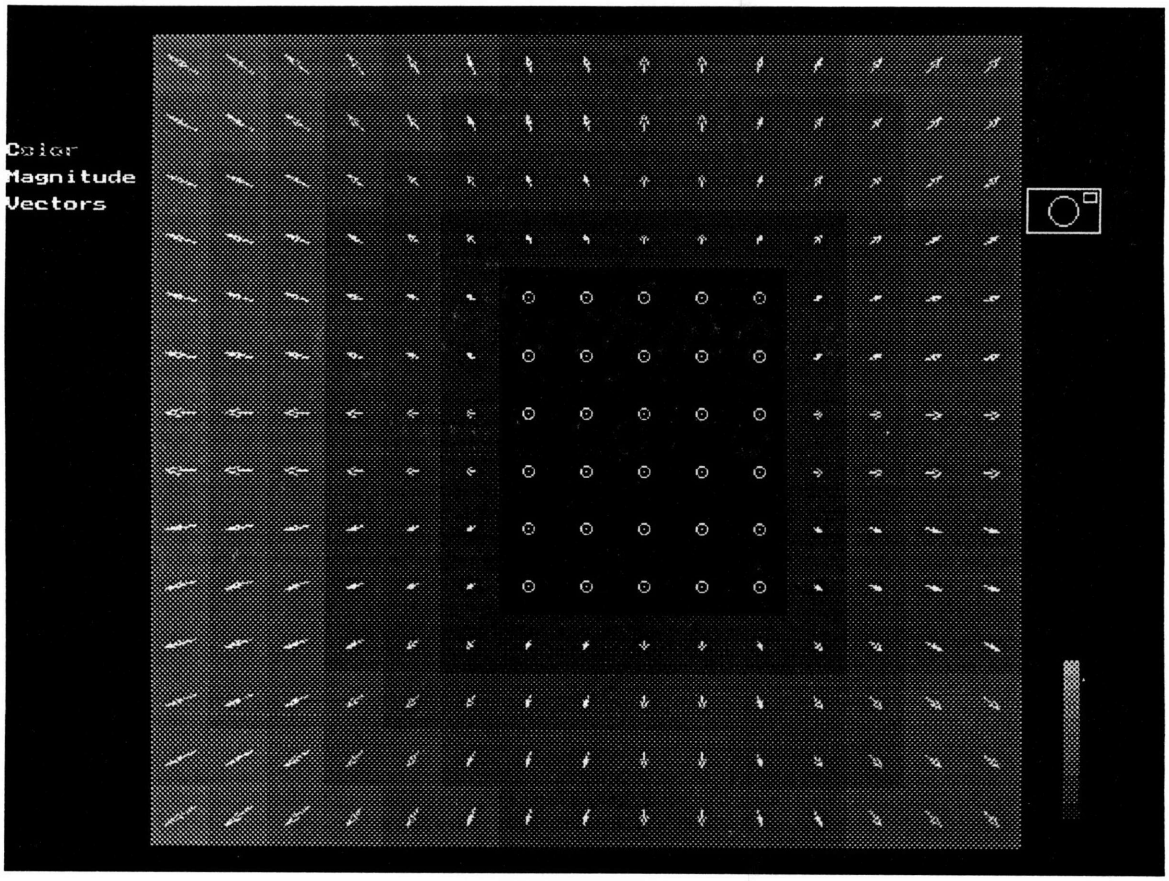


Figure 6.1 Flow Field Showing Motion Forward and to the Right

The small packaging size of the Vision System is 4.25"H *x* 5.0"W *x* 7.0"L, a total volume of 148.8 cubic inches, and fits directly into the assembly of the Companion wheel chair robot developed at the Intelligent Unmanned Vehicle Laboratory (IUVC) at the

Charles Stark Draper Laboratory in Cambridge Massachusetts. Using commercially available PC104 computer products and the Pattern Matching Module (PMM) described within this thesis it was possible to build this tiny Vision System.

The PMM is used to decrease the speed at which flow fields, like the flow field shown in Figure 6.1, are created and is constructed with a specialized correlator that contains 30 parallel processors. The PMM is designed to plug onto the PC104 standard bus and is addressable as a computer card. Traditionally, the computational bottleneck for many commercial Vision Systems has been the creation of flow fields. The PMM, however, was able to generate flow fields more quickly than it was able to process them. Based upon the PMM timing results, it is suggested that a suitable network of computers in conjunction with a single PMM could produce motion parameter information in real time. Further, approximately 75% of the delay in the PMM is due to memory bandwidth. Interconnecting the frame grabber to the PMM without the need for any bus transactions would decrease the flow field latency to a fraction of its value. The PMM fulfilled the goals of the hardware portion of this thesis.

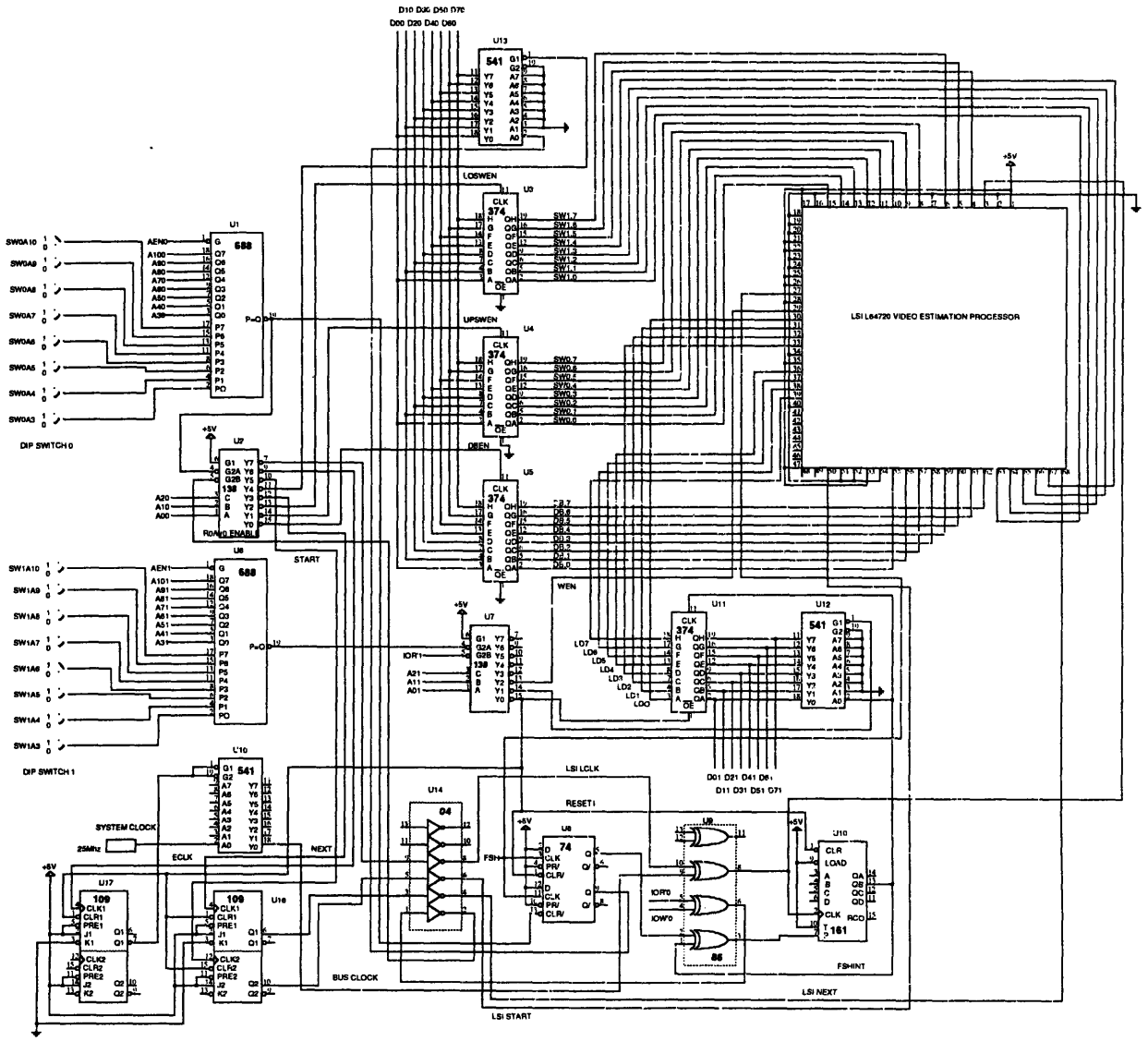
The goals of the software were to produce drivers for the PMM, a graphical user interface, and operational motion parameter computation code. The software was written in C and compiled optimizing for speed. The graphical User Interface was written by Ely Wilson; an example of the graphical User Interface is Figure 6.1. The PMM drivers were based upon existing C functions and a further optimization could be to write the PMM driver in a lower level code. It should be noted, however, that currently the PMM and its associated driver run faster than is necessary. The graphical user interface is an intuitive visualization of the flow field that displays the flow vectors and uses a red to blue or white to black color table to describe depth. The motion parameter computation code directly implements the iterative algorithm and is easily modified. A by-product of the software development is that simulations of every software and hardware component have been made. Currently, it is possible to run a complete simulation of the software on independent platforms that support DOS. The most important aspect of the Vision System software is that each process and function work.

The prototype Vision System can produce data every forty seconds, but with a slight upgrade it will produce data about every 500 milliseconds. The data that the Vision System produces is accurate to about 15% of the parameter under test. Overall, there are still small errors in the motion calculations and the simulated results indicate a reasonable improvement over the prototype. It is believed that most of the current errors are due to pixel resolution which can be solved by pixel interpolation. Pixel interpolation will produce a flow field where each flow vector will contain more significant digits. Using pixel interpolation can take the place of the currently used average vector elimination filter. Our camera geometry, however, is also to blame. Although, we measured the virtual pixel size, the measurement techniques could have been significantly improved using optics tables and more precise measurement equipment.

The software and hardware in this system are still under active research; this thesis will help shape future machine vision research by IUVC. One of the richest areas for future work is in algorithm development. In particular, the pattern matching algorithm could be enhanced to produce a more accurate flow field, which as we have already shown, will improve results. Another algorithm that could be improved is the motion parameter and depth map iteration loop; this loop requires a significant amount of time to converge and, on occasion, diverges.

The overall Vision System is a proof of the concept; it is small enough to be suitable for an autonomous application, potentially fast enough to aid in navigation, and, last but not least, it works!

APPENDIX A
PATTERN MATCHING MODULE SCHEMATIC



APPENDIX B
COMPLETE LISTING OF SOFTWARE

This code is the main loop that runs on the 386 processor.

```
#include <stdio.h>
#include <dos.h>
#include <conio.h>
#include <ctype.h>
#include <stdlib.h>
#include <math.h>

/*4/29/96 Anthony N. Lorusso
        This is the final code to generate a flow field and calculate
        motion from it using Ely Wilson's graphical interface.*/

int main(void){

/* Defines for system specific constants, this is to make the code
adaptable for future arrangements. */

#define VECTOR_NUM 210
#define VEC_COMP_NUM 420

MATRIX depth_map, motion, A, b;
char            InputC, Input_c,
                vect_comps[420];
unsigned char   * data_frame[14],
                * search_frame[14];
unsigned char   hardware_vector_array[VECTOR_NUM],
                filtered_vector_array[VECTOR_NUM];
int             ExitI,
                ValidArrayI;
double          convergence_tolerance,
                pixel_tolerance;

/*Beginning of code*/
depth_map = mat_creat(VECTOR_NUM, 1, UNDEFINED);
A = mat_creat(VEC_COMP_NUM, 6, UNDEFINED);
b = mat_creat(VEC_COMP_NUM, 1, UNDEFINED);

init_matrix(depth_map, VECTOR_NUM, 1, 1.0);          // Initialize depth map
init_frames(data_frame, search_frame);               // Initialize frames
initialize_LSI();                                    // Initialize LSI board
```

```
init_grabber();                                    //Initialize frame grabber

//Prompt for convergence tolerance
printf("What convergence tolerance do you want?\n");
scanf("\n%lf", &convergence_tolerance);

InitGVects();                                      // Initialize graphics display
ExitI = 0;
ValidArrayI = 0;
while (!ExitI){
        InitGVects();
        // Indicate ready to get first frame ...
        CameraOne();
        // Get keystroke
        InputC = getch();
        // Process keystroke ...
        switch (tolower(InputC)){
                case 'q':
                ExitI = 1;
                break;

                case ' ':

                // Get first frame
                CameraFlashOn();
                grab();
                CameraFlashOff();
                store_data_frame(data_frame);
                // Indicate ready for second frame
                CameraTwo();

                while (getch() != ' ');
                // Get second frame
                CameraFlashOn();
                grab();
                CameraFlashOff();
                store_search_frame(search_frame);
                // This compares the respective data blocks and search windows and
                // stores the results in vector_array
                generate_flow_field(search_frame, data_frame, hardware_vector_array);

                ValidArrayI = 1;
                GraphVects(hardware_vector_array);
                vector_array_to_components(vect_comps, hardware_vector_array);
                pixel_tolerance = 4.0; // Empirical flow field tolerance
```

```
                local_averaging_filter(vect_comps, pixel_tolerance);
                components_to_vector_array(vect_comps, filtered_vector_array);
                GraphVects(filtered_vector_array);
                ExitGVects();

                /* This calculates the six motion parameters.*/
                motion = calculate_motion(filtered_vector_array, depth_map,
                                                convergence_tolerance, A, b);
                printf("m=:\n");
                print_matrix(motion, 6, 1);
                printf("Hit any key to continue.\n");
                do {Input_c = getch();}
                while((tolower(Input_c) != ' ') && (tolower(Input_c) != 'q')
                        && (tolower(Input_c) != 'c'));
                break;

                case 't':
                InputC = getch();
                if (InputC >= 'a')
                        InputC -= 'a' - 10;
                else
                        InputC -= '0';

                hardware_vector_array[0] = InputC * 16;
                InputC = getch();
                if (InputC >= 'a')
                        InputC -= 'a' - 10;
                else
                        InputC -= '0';

                hardware_vector_array[0] += InputC;
                GraphVects(hardware_vector_array);
                ValidArrayI = 1;
                break;

                default:
                if (ConfigDisp(InputC) && ValidArrayI)
                        GraphVects(hardware_vector_array);
                        break;
                }
        }
ExitGVects();
return(0);
}
```

This is the file that contains all of the primary functions that are used

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <dos.h>
#include <conio.h>
/*#define DEBUG*/

void datablock_to_swindow(unsigned char *swindow, unsigned char *dblock, int xoff, int
yoff){
        int x, y;
        for(y = 0; y < 16; y++){
                for(x = 0; x < 16; x++){
                        swindow[xoff+x+y*32+yoff*32] = dblock[x+y*16];
                        }
                }
        }

void randomize_sw(unsigned char *swindow){
        #define MAX_RAND = 255;
        int x, y;
        for(y = 0; y < 32; y++){
                for(x = 0; x < 32; x++){
                        swindow[x+y*32] = rand();
                        }
                }
        }

void print_array(unsigned char *array, int col_limit, int row_limit){

        /* This function prints an arbitrary sized array. */

        int x, y, ct = 0;
        for(y = 0; y < row_limit; y++){
                for(x = 0; x < col_limit; x++){
                        printf(" %c", array[ct++]);
                        }
                printf("\n");
                }
        }

void print_hex_array(unsigned char *array, int col_limit, int row_limit){
```

```
        /* This function prints an arbitrary sized array. */

        int x, y, ct = 0;
        for(y = 0; y < row_limit; y++){
                for(x = 0; x < col_limit; x++){
                        printf(" %0x", array[ct++]);
                        }
                printf("\n");
                }
        }

void print_hex_unsigned_array(char *array, int col_limit, int row_limit){

        /* This function prints an arbitrary sized array. */

        int x, y, ct = 0;
        for(y = 0; y < row_limit; y++){
                for(x = 0; x < col_limit; x++){
                        printf(" %0x", array[ct++]);
                        }
                printf("\n");
                }
        }


void print_16_block(unsigned char *dblock){
        int x, y;
        for(y = 0; y < 16; y++){
                for(x = 0; x < 16; x++){
                        printf(" %c", dblock[x+y*16]);
                        }
                printf("\n");
                }
        }

void print_32_block(unsigned char *swindow){
        int x, y, ct = 0;
        for(y = 0; y < 32; y++){
                for(x = 0; x < 32; x++){
                        printf(" %c", swindow[ct++]);
                        }
                printf("\n");
                }
        }
```

```
void define_dblock(unsigned char *dblock){
        int pixelnum = 0;
        int x, y;
        for(y = 0; y < 16; y++){
                for(x = 0; x < 16; x++){
                        dblock[x+y*16] = pixelnum++;
                        }
                }
        }

void initialize_LSI(){
        int REST1 = 0x380, WEN = 0x382, ECLK = 0x396;
        inportb(REST1);
        inportb(ECLK);
        inportb(WEN);
        inportb(ECLK);
        inportb(REST1);
        }

void compare_db_to_sw(unsigned char *swindow, unsigned char *dblock, unsigned char
*distortion){
        int REST1 = 0x380, FSHCHK = 0x381, DBEN = 0x390,UPSWEN = 0x391,
                LOSWEN = 0x392, NEXT = 0x393, START = 0x395, LCLK = 0x397,
ECLK = 0x396;
        int i,j,k;
        unsigned char value;

        inportb(REST1);   /* reset output circuitry */
        /*inportb(WEN);*/    /* initialize LSI board */
        /*inportb(LCLK);*/
        inportb(START);
        inportb(LCLK);
        inportb(START);

        value = inportb(FSHCHK);
        while(value ==1){
                inportb(REST1);
                value = inportb(FSHCHK);
                }
        k=0;
        while( k != 2){
                j=0;
                while( j != 16){
                        i=0;
                        while( i != 16){
```

```
                        outportb(UPSWEN, swindow[j*32+i+16*k]);
                        outportb(LOSWEN, swindow[j*32+512+i+16*k]);
                        /*outportb(DBEN, dblock[j*16+i]);*/
                        #ifdef DEBUG
                        {
                        char cont = 'n';
                        printf("Upper byte (%d) =  %d\n", (i+16*k+j*32),
swindow[j*32+i+16*k]);
                        printf("Lower byte (%d) =  %d\n", (i+16*k+j*32+512),
swindow[j*32+512+i+16*k]);
                        printf("Data byte  (%d) =  %d\n", (i+j*16), dblock[j*16+i]);
                        while(cont != 'y'){
                                printf("Continue? y/n\n");
                                scanf("\n%c", &cont);
                                }
                        }
                        #endif
                        if( k == 1){
                                outportb(DBEN, dblock[j*16+i]);
                                }
                        inportb(NEXT);
                        inportb(LCLK);
                        inportb(NEXT);
                        inportb(LCLK);
                        i++;
                        }
                j++;
                }
        k++;
        if( k == 1){
                inportb(START);
                inportb(LCLK);
                inportb(START);
                inportb(LCLK);
                }
        }
        inportb(START);
        inportb(LCLK);
        inportb(START);
        inportb(LCLK);
        value = 0;
        i = 0;
        inportb(ECLK);
        while(value != 1){
                ++i;
```

```
                //inportb(LCLK);
                value = inportb(FSHCHK);
                if( i == 2400){
                        printf("TIMEOUT:\n");
                        inportb(ECLK);
                        inportb(REST1);
                        distortion[0] = 0xff;
                        }
                }
        //inportb(LCLK);
  inportb(ECLK);
        distortion[0] = inportb(REST1);
        /*value = inportb(REST1);*/
                 //for(i = 0; i < 6; ++i){
                /*value = inportb(REST1);*/
                //printf("%0x  ", inportb(REST1));
                //inportb(LCLK);
                //}
                 outportb(DBEN, 1);
                 //printf("\n");
}

void clear_db(unsigned char *dblock){
        int x,y;
        for(y = 0; y < 16; y++){
                for(x = 0; x < 16; x++){
                        dblock[x+y*16] = 48;
                        /* note that 48 is ASCII for 0, Borland seems to need this.*/
                        }
                }
        }

void clear_sw(unsigned char *swindow){
        int x,y;
        for(y = 0; y < 32; y++){
                for(x = 0; x < 32; x++){
                        swindow[x+y*32] = 48;
      /* note that 48 is ASCII for 0, Borland seems to need this.*/
                        }
                }
        }

void emulate_compare_db_to_sw(unsigned char *swindow, unsigned char *dblock,
unsigned char *e_distortion){
        /* simulates properly working pattern matching hardware */
```

```
        unsigned long int error, min_error, error_sum;
        unsigned int min_error_x, min_error_y, i, j, x, y;

        min_error_x = 0;
        min_error_y = 0;
        min_error = 600001;
        j = 0;
        while(j != 16){
                i = 0;
                while(i != 16){
                        error = 0;
                        error_sum = 0;
                        y = 0;
                        while(y != 16){
                                x=0;
                                while(x != 16){
                                        error = abs(swindow[x + i + y*32 +j*32] - dblock[x
+ y*16]);
                                        error_sum += error;
                                        ++x;
                                        }
                                ++y;
                                }
                        if(min_error >= error_sum){
                                min_error_x = i;
                                min_error_y = j;
                                min_error = error_sum;
                                }
                        if(error_sum >= 655361){
                                printf("error_sum has exceeded maximum value.\n");
                                }
                        ++i;
                        }
                ++j;
                }
        if(min_error_x <= 7){
                e_distortion[0] = (unsigned char)(min_error_x + 8);
                }
        else{
                e_distortion[0] = (unsigned char)(min_error_x - 8);
                }
        if(min_error_y <= 7){
                e_distortion[1] = (unsigned char)(min_error_y + 8);
                }
        else{
```

```
            e_distortion[1] = (unsigned char)(min_error_y - 8);
            }
        e_distortion[2] = (unsigned char)min_error;
        }


void fill_full_sw_buffer(unsigned char *swindow, unsigned char *search_strip,
                                                        int col_index, int
row_index){

/* This program is for the first LSI computation at the beginning of each
row, and to be used for every computation that is going to be emulated */
/* Also note that the computation number "comp_num" can be computed by
comp_num = col_index + row_index*32; */

            int i,j;
            i = 0;
            while(i != 32){
                    j = 0;
                    while(j != 32){
                            swindow[j+i*32] =
search_strip[j+i*256+col_index*16+row_index*16*256];
                            j++;
                            }
                    i++;
                    }
            }


void fill_half_sw_buffer(unsigned char *swindow, unsigned char *search_strip,
                                                        int col_index, int
row_index){

/* This program is for preparing the search window array for the repetetive
hardware computations in a single row. It loads the next half search window,
512 bytes, into the second half of the search window array. The array can then
be used to load the LSI hardware, and wait for a FSH. */

            int i,j;
            i = 0;
            while(i != 32){
                    j = 0;
                    while(j != 16){
```

```
                                swindow[16+j+i*32] =
search_strip[16+j+i*256+col_index*16+row_index*16*256];
                                j++;
                                }
                        i++;
                        }
                }

void fill_db_buffer(unsigned char *dblock, unsigned char *data_strip,
                                                        int col_index, int row_index){

/* This program is for all computationa using a data block */

        int i,j;
        i = 0;
        while(i != 16){
                j = 0;
                while(j != 16){
                        dblock[j+i*16] =
data_strip[(j+i*240+col_index*16+row_index*16*240)];
                        j++;
                        }
                i++;
                }
        }

void half_compare_db_to_sw(unsigned char *swindow, unsigned char *dblock, unsigned
char *distortion){

/* This program will load data fot the LSI hardware for computations after
the very first for every row. It will then return all computed values for
computations except the first and last in each row.*/

        int REST1 = 0x380, FSHCHK = 0x381, DBEN = 0x390,UPSWEN = 0x391,
                LOSWEN = 0x392, NEXT = 0x393, START = 0x395, LCLK = 0x397,
ECLK = 0x396;
        int i,j;
        unsigned char value;

        inportb(REST1);  /* Resets output circuitry. */
        inportb(START);  /* Resets LSI input controller.*/
        inportb(LCLK);
        inportb(START);
```

```
        inportb(LCLK);

        j=0;            /* Loads the half search window for the next computation.*/
        while( j != 16){
                i=0;
                while( i != 16){
                        outportb(UPSWEN, swindow[j*32+i+16]);
                        outportb(LOSWEN, swindow[j*32+512+i+16]);
                        outportb(DBEN, dblock[j*16+i]);
                        inportb(NEXT);
                        inportb(LCLK);
                        inportb(NEXT);
                        inportb(LCLK);
                        i++;
                        }
                j++;
                }
  inportb(REST1);  /* reset output circuitry */
        value = 0;
        i = 0;
        inportb(ECLK);   /* Turns LSI on board clock on.*/
        while(value != 1){
                ++i;
                value = inportb(FSHCHK);
                if( i == 2400){
                        printf("TIMEOUT:\n");
                        inportb(ECLK);   /*Turns LSI on board clock off.*/
                        inportb(REST1);
                        distortion[0] = 0xff;  /* default return value if TIMEOUT*/
                        }
                }
        if(value == 1){
                inportb(ECLK);      /*Turns LSI on board clock off.*/
                distortion[0] = inportb(REST1);  /*Returns computed value. */
                }
}

void last_compare_db_to_sw(unsigned char *distortion){

/* This program will retrieve data from the hardware for LSI
computations at the end of every row. */

        int REST1 = 0x380, FSHCHK = 0x381, START = 0x395, LCLK = 0x397, ECLK
= 0x396;
        int i;
```

```
        unsigned char value;

        inportb(REST1);  /* Reset output circuitry. */
        inportb(START);  /* Starts the final computation. */
        inportb(LCLK);
        inportb(START);
        inportb(LCLK);

        value = 0;
        i = 0;
        inportb(ECLK);   /* Turns LSI on board clock on.*/
        while(value != 1){
                ++i;
                value = inportb(FSHCHK);
                if( i == 2400){
                        printf("TIMEOUT:\n");
                        inportb(ECLK);     /* Turns LSI on board clock off.*/
                        inportb(REST1);
                        distortion[0] = 0xff; /*default return value if TIMEOUT */
                        }
                }
        if(value == 1){
                inportb(ECLK);       /* Turns LSI on board clock off.*/
                distortion[0] = inportb(REST1);  /*Returns computed value. */
                }
}



unsigned char ordered_to_bitwise(unsigned char *array){
        return((array[1] << 4) | array[0]);
        }

void bitwise_to_ordered(unsigned char *array, unsigned char byte){
        array[1] = ((byte >> 4) & 0x0f);
        array[0] = byte & 0x0f;
        }


void fill_strip(unsigned char *strip, int length, unsigned char fill_value){

        /* This function places a given number of specified characters into a strip
        for testing purposes. */

        int i =0;
```

```
        while(i != length){
                strip[i] = fill_value;
                i++;
                }
        }

void emulate_compare_strips(unsigned char *search_strip, unsigned char *data_strip,
                                                                unsigned char
*dblock, unsigned char *swindow,
                                                                unsigned char
*vector_array, int strip_num){

        /* This function takes two strips, a data_strip, and a search_strip, and
        computes 15 flow vectors from them and stores them in the vector_array */

        int i;
        unsigned char e_distortion[3];
        i = 0;
        while(i != 15){
                fill_db_buffer(dblock, data_strip, i, 0);
                fill_full_sw_buffer(swindow, search_strip, i, 0);
                /*printf("This is the emulator data block.\n");
                print_hex_array(dblock, 16, 16);
                printf("this is the emulator search window.\n");
                print_hex_array(swindow, 32, 32);
                printf("Hit a key to continue.\n");
                getch(); */

                /* Note that this function will need to be changed to incorporate
                the half search window dowload required by the LSI chip.*/

                emulate_compare_db_to_sw(swindow, dblock, e_distortion);
                //printf("Our emulator vector is %0x\n", ordered_to_bitwise(e_distortion));
                vector_array[i+strip_num*15] = ordered_to_bitwise(e_distortion);
                i++;
                }
        }

void compare_strips(unsigned char *search_strip, unsigned char *data_strip,
                                                        unsigned char *dblock, unsigned char
*swindow,
                                                        unsigned char *vector_array, int
strip_num){

        /* This function takes two strips, a data_strip, and a search_strip, and
```

```
        computes 15 flow vectors from them and stores them in the vector_array */

        int i;
        unsigned char e_distortion[3];
        i = 0;
        while(i != 15){
                fill_db_buffer(dblock, data_strip, i, 0);
                fill_full_sw_buffer(swindow, search_strip, i, 0);
                /*printf("This is the hardware data block.\n");
                print_hex_array(dblock, 16, 16);
                printf("this is the hardware search window.\n");
                print_hex_array(swindow, 32, 32);
                printf("Hit a key to continue.\n");
                getch(); */

                /* Note that this function will need to be changed to incorporate
                the half search window dowload required by the LSI chip.*/

                compare_db_to_sw(swindow, dblock, e_distortion);
                //printf("Our hardware vector is %0x\n", e_distortion[0]);
                vector_array[i+strip_num*15] = e_distortion[0];
                i++;
                }
        }

void clear_data_strip(unsigned char *data_strip){
        int x,y;
        for(y = 0; y < 16; y++){
                for(x = 0; x < 240; x++){
                        data_strip[(x+y*240)] = 48;
                        /* note that 48 is ASCII for 0, Borland seems to need this.*/
                        }
                }
        }

void clear_search_strip(unsigned char *search_strip){
        int x,y;
        for(y = 0; y < 32; y++){
                for(x = 0; x < 256; x++){
                        search_strip[(x+y*256)] = 48;
                        /* note that 48 is ASCII for 0, Borland seems to need this.*/
                        }
                }
        }
```

```
void init_frames(unsigned char **data_frame, unsigned char **search_frame){

/* The following are the frame initializations */

int z;
for(z = 0; z < 14; z++){
        data_frame[z] = (unsigned char *)malloc(3840*sizeof(unsigned char));
        if(data_frame[z] == NULL){
                printf("Cannot allocate data_frame.\n");
                exit(1);
                }
        search_frame[z] = (unsigned char*)malloc(8192*sizeof(unsigned char));
        if(search_frame[z] == NULL){
                printf("Cannot allocate search_frame.\n");
                exit(1);
                }
        clear_data_strip(data_frame[z]);
        clear_search_strip(search_frame[z]);
        }
}

void init_grabber(void){

/* The following is the frame grabber initialization */

char *err;

        err = init_library();
        if(err != NULL) {
                puts(err);
                exit(1);
        }
/* Set user options */

low_res;
set_page(0);
}

void full_compare_db_to_sw(unsigned char *swindow, unsigned char *dblock,
                                                        unsigned char
*next_half_swindow, unsigned char *next_dblock,
                                                        unsigned char
*distortion){

        int REST1 = 0x380, FSHCHK = 0x381, DBEN = 0x390,UPSWEN = 0x391,
```

```
        LOSWEN = 0x392, NEXT = 0x393, START = 0x395, LCLK = 0x397,
ECLK = 0x396;
    int i,j,k;
    unsigned char value;

    inportb(REST1);  /* Reset output circuitry. */
    inportb(START);  /* Reset the LSI input controller. */
    inportb(LCLK);
    inportb(START);
    inportb(LCLK);

    k=0;          /* Load the first computation's data. */
    while( k != 2){
        j=0;
        while( j != 16){
            i=0;
            while( i != 16){
            outportb(UPSWEN, swindow[j*32+i+16*k]);
            outportb(LOSWEN, swindow[j*32+512+i+16*k]);
            if( k == 1){
                outportb(DBEN, dblock[j*16+i]);
                }
            inportb(NEXT);
            inportb(LCLK);
            inportb(NEXT);
            inportb(LCLK);
            i++;
            }
        j++;
        }
    k++;
    if( k == 1){     /* Reset the input controller. */
        inportb(START);
        inportb(LCLK);
        inportb(START);
        inportb(LCLK);
        }
    }
    inportb(START);      /* These lines start the computation, */
    inportb(LCLK);   /* and they start swapping the input buffers. */
    inportb(START);
    inportb(LCLK);

    /* These lines load the next half search window. */
```

```
        j=0;
        while( j != 16){
                i=0;
                while( i != 16){
                        outportb(UPSWEN, next_half_swindow[j*32+i+16]);
                        outportb(LOSWEN, next_half_swindow[j*32+512+i+16]);
                        outportb(DBEN, next_dblock[j*16+i]);
                        inportb(NEXT);
                        inportb(LCLK);
                        inportb(NEXT);
                        inportb(LCLK);
                        i++;
                }
        j++;
        }

        value = 0;
        i = 0;
        inportb(ECLK);          /* This starts the on board LSI clock. */
        while(value != 1){
                ++i;
                value = inportb(FSHCHK);
                if( i == 2400){
                        printf("TIMEOUT:\n");
                        inportb(ECLK); /* This turns off the on board LSI clock. */
                        inportb(REST1);
                        distortion[0] = 0xff;  /* default return value if TIMEOUT */
                        }
                }
        inportb(ECLK);                          /* This turns off the on board LSI clock. */
        distortion[0] = inportb(REST1);   /* actual return value */
}


void optimized_compare_strips(unsigned char *search_strip, unsigned char *data_strip,
                                                        unsigned char *dblock, unsigned char
*swindow, unsigned char *next_half_swindow,
                                                        unsigned char *next_dblock,
unsigned char *vector_array, int strip_num){

        /* This function takes two strips, a data_strip, and a search_strip, and
        computes 15 flow vectors from them and stores them in the vector_array */

        int i;
        unsigned char e_distortion[3];
```

```
        i = 0;
        /*clear_sw(swindow);
        clear_sw(next_half_swindow);
        clear_db(dblock);
        clear_db(next_dblock);*/

        while(i != 15){
                /* These next three functions should be removed! */
                if(i == 0){
                        fill_db_buffer(dblock, data_strip, i, 0);
                        fill_db_buffer(next_dblock, data_strip, (i+1), 0);
                        fill_full_sw_buffer(swindow, search_strip, i, 0);
                        fill_half_sw_buffer(next_half_swindow, search_strip, (i+1),0);
                        full_compare_db_to_sw(swindow, dblock, next_half_swindow,
next_dblock, e_distortion);
                        }
          else if(i > 0 && i < 14){
                        fill_db_buffer(next_dblock, data_strip, (i+1), 0);
                        fill_half_sw_buffer(next_half_swindow, search_strip, (i+1), 0);
                        half_compare_db_to_sw(next_half_swindow, next_dblock,
e_distortion);
                        }
                else{
                        last_compare_db_to_sw(e_distortion);
                        }
                vector_array[i+strip_num*15] = e_distortion[0];
                i++;
                }
        }
```

This is the primary file for the user interface. The user interface is a graphical display that was developed by Ely Wilson. All of the following code is or the graphical interface.

```
#include "graphvec.h"
#include <conio.h>
#include <ctype.h>
#include <graphics.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
```

```
struct
{
        int ColorI,
                MagnitudeI,
                VectorsI;
} DispConfigT;

int ConfigDisp(char InputC)
{
        void MakeScale();
        int RetValI;

        // Return TRUE if display redraw is necessary
        switch (tolower(InputC))
        {
        case 'c':
                DispConfigT.ColorI = !DispConfigT.ColorI;
                settextjustify(LEFT_TEXT, CENTER_TEXT);
                setcolor(15);
                moveto(0, COLOR_COORD);
                outtext("C");
                if (!DispConfigT.ColorI)
                        setcolor(14);
                outtext("olor");
                MakeScale();
                RetValI = 0;
                break;

        case 'm':
                DispConfigT.MagnitudeI = !DispConfigT.MagnitudeI;
                settextjustify(LEFT_TEXT, CENTER_TEXT);
                setcolor(15);
                moveto(0, MAG_COORD);
                outtext("M");
                if (!DispConfigT.MagnitudeI)
                        setcolor(14);
                outtext("agnitude");
                RetValI = 1;
                break;

        case 'v':
                DispConfigT.VectorsI = !DispConfigT.VectorsI;
                settextjustify(LEFT_TEXT, CENTER_TEXT);
                setcolor(15);
                moveto(0, VECT_COORD);
```

```
                outtext("V");
                if (!DispConfigT.VectorsI)
                        setcolor(14);
                outtext("ectors");
                RetValI = 1;
                break;

        default:
                RetValI = 0;
                break;
        }
return(RetValI);
}


void InitGVects()
{
        void DrawCamera(int, int),
                MakeScale();
        int ColorI,
                GDriverI,
                GModeI,
                XCoordI,
                YCoordI,
                PointsAI[10],
                ErrorCode;

        // Load driver
        GDriverI = VGA;
        GModeI = VGAHI;
        initgraph(&GDriverI, &GModeI, 0);
        ErrorCode = graphresult();
        printf("initgraph returned %d\n", ErrorCode);

        setlinestyle(SOLID_LINE, 0, NORM_WIDTH);

        // Set default values
        DispConfigT.ColorI = 1;
        DispConfigT.MagnitudeI = 1;
        DispConfigT.VectorsI = 1;

        // Gray
        setpalette(14, 14);
        setrgbpalette(14, 35, 35, 35);
```

```
        // White
        setpalette(15, 15);
        setrgbpalette(15, 63, 63, 63);

        // Draw camera
        DrawCamera(LEFT_MARGIN + REGION_WIDTH * 15.75,
                                TOP_MARGIN + REGION_HEIGHT * 3);

        settextjustify(LEFT_TEXT, CENTER_TEXT);
        setcolor(15);
        outtextxy(0, COLOR_COORD, "Color");
        outtextxy(0, MAG_COORD, "Magnitude");
        outtextxy(0, VECT_COORD, "Vectors");

        MakeScale();

        // Draw scale
        XCoordI = LEFT_MARGIN + REGION_WIDTH * 15.75;
        YCoordI = TOP_MARGIN + REGION_HEIGHT * 14 - 8 * 13;

        for (ColorI = 13; ColorI > 0; ColorI--, YCoordI += 8)
        {
                PointsAI[0] = XCoordI;
                PointsAI[1] = YCoordI;
                PointsAI[2] = XCoordI + 8;
                PointsAI[3] = YCoordI;
                PointsAI[4] = XCoordI + 8;
                PointsAI[5] = YCoordI + 8;
                PointsAI[6] = XCoordI;
                PointsAI[7] = YCoordI + 8;
                PointsAI[8] = XCoordI;
                PointsAI[9] = YCoordI;

                setcolor(ColorI);
                setfillstyle(SOLID_FILL, ColorI);

                fillpoly(5, PointsAI);
        }
return;
}


void MakeScale()
{
        int ColorI,
```

```
                RedI;
        if (DispConfigT.ColorI)
                // Set up color palette
                for (ColorI = 1, RedI = 0; ColorI < 14; ColorI++, RedI += 63/13)
                {
                        setpalette(ColorI, ColorI);
                        setrgbpalette(ColorI, RedI, 0, (63 - RedI) / 1.5);
                }
        else
                // Set up grayscale palette
                for (ColorI = 1, RedI = 0; ColorI < 14; ColorI++, RedI += 63/13)
                {
                        setpalette(ColorI, ColorI);
                        setrgbpalette(ColorI, RedI/1.2, RedI/1.2, RedI/1.2);
                }
return;
}


void ExitGVects()
{
        restorecrtmode();
}


void GraphVects(unsigned char VectsAUC[210])
{
        void WriteVector(int, int, int, int, int);

        int ColI,
                RowI,
                VectMagI,
                XDistI,
                YDistI,
                PointsAI[10];

        // Set initial Y coordinates
        PointsAI[1] = TOP_MARGIN;
        PointsAI[3] = TOP_MARGIN;
        PointsAI[5] = TOP_MARGIN + REGION_WIDTH;
        PointsAI[7] = TOP_MARGIN + REGION_WIDTH;
        PointsAI[9] = TOP_MARGIN;

        // Begin drawing display ...
        for (RowI = 0; RowI < 14; RowI++)
```

```
{
        // Reset X coordinates
        PointsAI[0] = LEFT_MARGIN;
        PointsAI[2] = LEFT_MARGIN + REGION_WIDTH;
        PointsAI[4] = LEFT_MARGIN + REGION_WIDTH;
        PointsAI[6] = LEFT_MARGIN;
        PointsAI[8] = LEFT_MARGIN;

        for (ColI = 0; ColI < 15; ColI++)
        {
                // Get X and Y change
                XDistI = (VectsAUC[RowI * 15 + ColI] & 0x0F);
                if (XDistI > 7)

                        XDistI -= 16;

                YDistI = (VectsAUC[RowI * 15 + ColI] >> 4);
                if (YDistI > 7)

                        YDistI -= 16;

                // Calculate vector magnitude
                VectMagI = sqrt(pow(XDistI, 2) + pow(YDistI, 2));

                // Fill region
                setcolor(VectMagI + 1);
                setfillstyle(SOLID_FILL, VectMagI + 1);
                fillpoly(5, PointsAI);

                // Annotate region
                WriteVector(PointsAI[0] + REGION_WIDTH/2, PointsAI[1] +
                                REGION_WIDTH/2,
                                XDistI, YDistI, VectMagI);

                // Increment X coordinates
                PointsAI[0] += REGION_WIDTH;
                PointsAI[2] += REGION_WIDTH;
                PointsAI[4] += REGION_WIDTH;
                PointsAI[6] += REGION_WIDTH;
                PointsAI[8] += REGION_WIDTH;
        }

        // Increment Y coordinates
        PointsAI[1] += REGION_HEIGHT;
        PointsAI[3] += REGION_HEIGHT;
```

```
                PointsAI[5] += REGION_HEIGHT;
                PointsAI[7] += REGION_HEIGHT;
                PointsAI[9] += REGION_HEIGHT;
        }
return;
}


void WriteVector(int XCoordI, int YCoordI, int XDistI, int YDistI, int VectMagI)
{
        char  OutStringAC[10];
        int   XNormI,
                        YNormI;
        double AngleD;
        setcolor(15);
        if (DispConfigT.VectorsI)
                if (VectMagI < 2)
                {
                        putpixel(XCoordI, YCoordI, 15);
                        circle(XCoordI, YCoordI, 3);
                }
                else
                {
                        // "Normalize" distances
                        AngleD = atan2((double)YDistI, (double)XDistI);

                        XNormI = 8 * cos(AngleD);
                        YNormI = 8 * sin(AngleD);
                        if (!DispConfigT.MagnitudeI)
                        {
                                XDistI = XNormI;
                                YDistI = YNormI;
                        }

                        // Draw arrow
                        line(XCoordI + YNormI / 4, YCoordI - XNormI / 4, XCoordI +
                                XDistI,
                                  YCoordI + YDistI);
                        line(XCoordI - YNormI / 4, YCoordI + XNormI / 4, XCoordI +
                                XDistI,
                                  YCoordI + YDistI);
                        line(XCoordI, YCoordI, XCoordI - XDistI, YCoordI - YDistI);
                }

        else if (DispConfigT.MagnitudeI)
```

```
        {
                settextjustify(CENTER_TEXT, CENTER_TEXT);
                sprintf(OutStringAC, "%d", VectMagI);
                outtextxy(XCoordI, YCoordI, OutStringAC);
        }
return;
}


int CameraXI,
        CameraYI,
        FlashAI[10];

void DrawCamera(int XCoordI, int YCoordI)
{
        int PointsAI[10];
        setcolor(15);
        setfillstyle(SOLID_FILL, 0);

        // Draw camera body
        PointsAI[0] = XCoordI - CAMERA_WIDTH/2;
        PointsAI[1] = YCoordI - CAMERA_HEIGHT/2;
        PointsAI[2] = XCoordI + CAMERA_WIDTH/2;
        PointsAI[3] = YCoordI - CAMERA_HEIGHT/2;
        PointsAI[4] = XCoordI + CAMERA_WIDTH/2;
        PointsAI[5] = YCoordI + CAMERA_HEIGHT/2;
        PointsAI[6] = XCoordI - CAMERA_WIDTH/2;
        PointsAI[7] = YCoordI + CAMERA_HEIGHT/2;
        PointsAI[8] = PointsAI[0];
        PointsAI[9] = PointsAI[1];

        fillpoly(5, PointsAI);

        // Draw camera lens
        fillellipse(XCoordI, YCoordI, CAMERA_HEIGHT /3, CAMERA_HEIGHT / 3);

        // Draw flash
        FlashAI[0] = XCoordI + CAMERA_WIDTH/2 - FLASH_WIDTH;
        FlashAI[1] = YCoordI - CAMERA_HEIGHT/2 + 2;
        FlashAI[2] = XCoordI + CAMERA_WIDTH/2 - 2;
        FlashAI[3] = YCoordI - CAMERA_HEIGHT/2 + 2;
        FlashAI[4] = XCoordI + CAMERA_WIDTH/2 - 2;
        FlashAI[5] = YCoordI - CAMERA_HEIGHT/2 + FLASH_HEIGHT;
        FlashAI[6] = XCoordI + CAMERA_WIDTH/2 - FLASH_WIDTH;
```

```
        FlashAI[7] = YCoordI - CAMERA_HEIGHT/2 + FLASH_HEIGHT;
        FlashAI[8] = FlashAI[0];
        FlashAI[9] = FlashAI[1];

        fillpoly(5, FlashAI);

        // Record camera location
        CameraXI = XCoordI;
        CameraYI = YCoordI;
        return;
}


void CameraOne()
{
        void CameraNone();
        CameraNone();
        settextjustify(CENTER_TEXT, CENTER_TEXT);
        outtextxy(CameraXI, CameraYI, "1");
}


void CameraTwo()
{
        void CameraNone();

        CameraNone();
  settextjustify(CENTER_TEXT, CENTER_TEXT);
        outtextxy(CameraXI, CameraYI, "2");
}


void CameraNone()
{
        setcolor(15);
        setfillstyle(SOLID_FILL, 0);
        fillellipse(CameraXI, CameraYI, CAMERA_HEIGHT / 3, CAMERA_HEIGHT /
                        3);
}


void CameraFlashOn()
{
        setcolor(15);
        setfillstyle(SOLID_FILL, 15);
```

```
        fillpoly(5, FlashAI);
}


void CameraFlashOff()
{
        setcolor(15);
        setfillstyle(SOLID_FILL, 0);
        fillpoly(5, FlashAI);
}
```

This is the code that calculates motion

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

/* 5/14/96 Anthony N. Lorusso
        This file contains all of the necessary functions to calculate the depth
        map and six motion parameters from a motion field of 210 flow vectors. */

void fill_A_matrix(MATRIX A, MATRIX depth_map){

/* A is a 420X6 matrix. This function uses the depth map and fills the
        elements of A with the correct values from the Longuet Higgins and
        Pradzny equations. This is to solve Am=b.*/

        /* environment scale factors in meters*/
        #define focal_length 8.5e-3
        #define x_data_block_length 1.0e-3
        #define y_data_block_length 8.0e-4
        #define pixels_per_x 16
        #define pixels_per_y 16
        #define scale_factor_x 62.5e-6  //6e-5
        #define scale_factor_y 50.0e-6  //6e-5

        int i, j;
        double x, y, x_scaled, y_scaled, Z;

        /* if "i" is odd then fill with the second row sequence, otherwise fill
        with the first row sequence. */

        i = 0;
```

```
j = 0;

/*Set the x & y values to their initial relative positions on the image
plane. Scaled x & y are used to accurately reflect the placement of the
coordinates with respect to the pixels generated by the frame grabber.
This is so the u & v components will actually be correct proportioinal
to the x & y positions. */

x = -7.0;
y = 6.5;

while(i != 420){
        Z = depth_map[j][0];

        /* multiplying by the scale factor and dividing by the displacements
        accounts for real world displacements*/

        y_scaled = (y*y_data_block_length/focal_length);
        x_scaled = (x*x_data_block_length/focal_length);

        if(i%2 == 0){
                A[i][0] = -(1/Z);
                A[i][1] = 0.0;
                A[i][2] = (x_scaled/Z);
                A[i][3] = (x_scaled*y_scaled);
                A[i][4] = -(x_scaled*x_scaled+1);
                A[i][5] = y_scaled;
                i++;
                }
        else{
                A[i][0] = 0.0;
                A[i][1] = -(1/Z);
                A[i][2] = (y_scaled/Z);
                A[i][3] = (y_scaled*y_scaled+1);
                A[i][4] = -(x_scaled*y_scaled);
                A[i][5] = -x_scaled;
                j++;
                x++;
                i++;
                }

        if(x >= 8.0){
                x = -7.0;
                y--;
                }
```

```
            }
        }

void vector_array_to_components(char *vect_comps,
                                unsigned char *vector_array){

/*This function will take a vector_array of 210 vectors created by the
hardware and seperate the vectors into their u&v components, and then store
them in a component array, vect_comps, of 420 components.*/

        unsigned char flow_vector[2];
        int i, u, v;

        i=0;
        while(i != 210){
                bitwise_to_ordered(flow_vector, vector_array[i]);

                u = flow_vector[0];             /* u component, x axis */
                if(u > 7){                      /* conversion for hardware */
                        u -= 16;}
                v = flow_vector[1];             /* v component, y axis */
                if(v > 7){                      /* conversion for hardware */
                        v -= 16;}
                vect_comps[(i*2)] = u;
                vect_comps[(i*2+1)] = -v;
                i++;
        }
}

void components_to_vector_array(char *vect_comps, unsigned char *vector_array){

/*This function will take a "vect_comps" matrix of 420 u&v components and recombine
the vector components so that they can be displayed with GraphVects().*/

        unsigned char flow_vector[2];
        int i, u, v;

        i=0;
        while(i != 210){
                u = vect_comps[(i*2)];
                v = -vect_comps[(i*2+1)];

                if(u < 0)                       /* conversion for hardware */
                        u += 16;
```

```
                if(v < 0)                       /* conversion for hardware */
                        v += 16;

                flow_vector[0] = (unsigned char) u;
                flow_vector[1] = (unsigned char) v;

                vector_array[i] = ordered_to_bitwise(flow_vector);
                i++;
        }
}

void get_filter_data(int *filter_data, int vector_num,
                                                        char *vect_comps){
/*This function loads the correct data structure with the current vector
information so that this single vector can be filtered.*/

        int vector_num_up, vector_num_down, vector_num_left, vector_num_right;

/*get information for the flow vector above the one being filtered. */

        vector_num_up = vector_num - 15;
        if(vector_num_up < 0){
                filter_data[0] = 0;
                filter_data[1] = 0;
                }
        else{
                filter_data[0] = (int)vect_comps[vector_num_up*2];
                filter_data[1] = (int)vect_comps[vector_num_up*2+1];
                }

/*get information for the flow vector below the one being filtered. */

        vector_num_down = vector_num + 15;
        if(vector_num_down > 209){
                filter_data[2] = 0;
                filter_data[3] = 0;
                }
        else{
                filter_data[2] = (int)vect_comps[vector_num_down*2];
                filter_data[3] = (int)vect_comps[vector_num_down*2+1];
                }

/*get information for the flow vector to the left of the one being filtered. */

        vector_num_left = vector_num -1;
```

```
        if((vector_num % 15) == 0){
                filter_data[4] = 0;
                filter_data[5] = 0;
                }
        else{
                filter_data[4] = (int)vect_comps[vector_num_left*2];
                filter_data[5] = (int)vect_comps[vector_num_left*2+1];
                }
/*get information for the flow vector to the right of the one being filtered. */

        vector_num_right = vector_num +1;
        if((vector_num % 14) == 0){
                filter_data[6] = 0;
                filter_data[7] = 0;
                }
        else{
                filter_data[6] = (int)vect_comps[vector_num_right*2];
                filter_data[7] = (int)vect_comps[vector_num_right*2+1];
                }
        }

void local_averaging_filter(char *vect_comps, double pixel_tolerance){

/*This function applies local averaging to the vector flow field in order to
eliminate erroneous flow vectors.*/

        int i,u, v, filter_data[8];
        double ave_u, ave_v, abs_diff_u, abs_diff_v;

/*main loop*/

        for(i = 0; i < 210; i++){
                get_filter_data(filter_data, i, vect_comps);
                ave_u = (filter_data[0]+filter_data[2]+
                                        filter_data[4]+filter_data[6])/4;
                ave_v = (filter_data[1]+filter_data[3]+
                                        filter_data[5]+filter_data[7])/4;
                u = (int)vect_comps[i*2];
                v = (int)vect_comps[i*2+1];

                abs_diff_u = abs(ave_u - u);
                abs_diff_v = abs(ave_v - v);

                if(abs_diff_u > pixel_tolerance ||
                        abs_diff_v > pixel_tolerance){
```

```
                    vect_comps[i*2] = 0;
                    vect_comps[i*2+1] = 0;
                }
            }
        }

void fill_b_and_depth_matrix( MATRIX b, MATRIX depth_map, unsigned char
*vector_array){

/* This function will take the vector_array of 210 vectors from the hardware
and seperate the vectors into their u&v components, and then store them in a
component array, b, of 420 components. The component array, however, is also
scaled to reflect the calibration of the system.After the component array has
been generated, the depth map is filled using the components.*/

    #define infinite_depth 1000
    #define minimum_depth 1e-6
    #define scale_factor_x 62.5e-6
    #define scale_factor_y 50.0e-6

    unsigned char flow_vector[2];
    int i;
    double u, v/*, vector_magnitude, Z*/;

    i=0;
    while(i != 210){
        bitwise_to_ordered(flow_vector, vector_array[i]);

        u = flow_vector[0];             /* u component, x axis */
        if(u > 7)                                   /* conversion for hardware */
            u -= 16;

        v = flow_vector[1];     /* v component, y axis */
        if(v > 7)           /* conversion for hardware */
            v -= 16;

        /*multiplying by the scale factor and dividing by the focal length
        account for the real world displacements */

        b[(i*2)][0] = (u*scale_factor_x/focal_length);
        b[(i*2+1)][0] = (-v*scale_factor_y/focal_length);

        else if(Z > infinite_depth)
            depth_map[i][0] = infinite_depth;
```

```
            i++;
            }
        }

double depth_calculation(double x, double y, double u, double v,
                                MATRIX m){

/* This function takes all of the necessary components to compute an new
value for depth, and returns it. */

        #define infinite_depth 1000
        #define minimum_depth 1e-6

        double U, V, W, A, B, C, Su, Sv, Z;

        U = m[0][0];
        V = m[1][0];
        W = m[2][0];
        A = m[3][0];
        B = m[4][0];
        C = m[5][0];

        Su = ((-U)+(x*W));
        Sv = ((-V)+(y*W));

        /* calculate depth */
        Z = ((Su*Su + Sv*Sv)/
                        (((u-(x*y*A-(x*x+1)*B+y*C))*Su)+(v-((y*y+1)*A-(x*y*B)-
                        (x*C)))*Sv));

        if(Z > infinite_depth)
                Z = infinite_depth;
        if( Z == 0.0)
                Z = 0.00000001; //this number seems smaller than most depths.
                return(Z);
        }

double update_depth_map( MATRIX depth_map,  MATRIX m,  MATRIX b){

/* This function takes the depth map and a new set of motion parameters,
compares each new depth value with the old one, replaces the old depth value,
and returns the largest depth difference (dpeth_convergence). */

        #define pixels_per_x 16
        #define pixels_per_y 16
```

```
#define x_data_block_length 1.0e-3   //9.6e-4
#define y_data_block_length 8.0e-4   //9.6e-4
#define scale_factor_x 62.5e-6       //6e-5
#define scale_factor_y 50.0e-6       //6e-5

double Z, depth_convergence, old_Z, abs_depth_difference,
                x, y, x_scaled, y_scaled, u, v;
int i;

x = -7.0;
y = 6.5;
i = 0;
depth_convergence = 0;
while(i != 210){

        /*multiplying by the scale factor and dividing by the focal length
        account for real world displacements*/

        x_scaled = (x*x_data_block_length/focal_length);
        y_scaled = (y*y_data_block_length/focal_length);
        old_Z = depth_map[i][0];
        u = b[(i*2)][0];
        v = b[(i*2+1)][0];
        Z = depth_calculation(x_scaled, y_scaled, u, v, m);
        abs_depth_difference = fabs(old_Z - Z);
        if(abs_depth_difference > depth_convergence){
                depth_convergence = abs_depth_difference;
                }
        depth_map[i][0] = Z;
        i++;
        x++;
        if(x >= 8.0){
                x = -7.0;
                y--;
                }
        }
return(depth_convergence);
}

void print_matrix( MATRIX MATRIX, int row_num, int col_num){

/* This function prints an arbitrary size matrix of doubles. The function
print_array, or print_hex_array, will print an arbitrary size array of
unsigned characters. */
```

```
        int x,y;
        for(y = 0; y < row_num; y++){
                for(x = 0; x < col_num; x++){
                        printf(" %lf", MATRIX[y][x]);
                        }
                printf("\n");
                }
        }

void init_matrix(MATRIX MATRIX, int row_num, int col_num, double value){

/* This function initializes an arbitrary size matrix of doubles, it fills
every element in the matrix with the double "value." The functions
print_array, or print_hex_array, will print an arbitrary size array of
unsigned characters. */

        int x,y;
        for(y = 0; y < row_num; y++){
                for(x = 0; x < col_num; x++){
                        MATRIX[y][x] = value;
                        }
                }
        }

MATRIX calculate_motion(unsigned char *vector_array, MATRIX depth_map,
                                                        double
convergence_tolerance, MATRIX A, MATRIX b){

/* This function calculates the six motion parameters of motion and the
depth map. Currently, it only returns the six parameters of motion. */

/* Defines for system specific constants, this is to make the code
adaptable for future arrangements. */

#define VECTOR_NUM 210
#define VEC_COMP_NUM 420

MATRIX A_trans;
double depth_convergence;

/* create the matrices */
m = mat_creat(6, 1, UNDEFINED);

/* fill matrices */
fill_b_and_depth_matrix(b, depth_map, vector_array);
```

```
init_matrix(m, 6, 1, .05);
init_matrix(depth_map, 210, 1, 1);
depth_convergence = update_depth_map(depth_map, m, b);
fill_A_matrix(A, depth_map);
A_trans = mat_tran(A);

/* complete depth map convergence loop */
while(depth_convergence > convergence_tolerance){
        printf("depth map convergence = %lf\n", depth_convergence);
        print_matrix(m, 6, 1);
        /* calculate motion */
        m = mat_mul(mat_inv(mat_mul(A_trans, A)), mat_mul(A_trans, b));
        /* update depth map */
        depth_convergence = update_depth_map(depth_map, m, b);
        /* make the next A & A_trans matrices */
        fill_A_matrix(A, depth_map);
        A_trans = mat_tran(A);
        }

/* return motion parameters */
return(m);
}
```

The code that performs the matrix functions is not included here. The unmodified code was written by Patrick KO shu pui and is available on the world wide web for educational use only.

REFERENCES

[1] Gilbert Strang, Linear Algebra and its Applications, Harcourt Brace Jovanovich, 1988, pp. 167

[2] The Chemical Rubber Company, CRC Press, Standard Mathematical Tables, 28th. Edition, pp. 297.

[3] Berthold Klaus Paul Horn, personal coversation during 3/96

[4] Longuet Higgins H.C., & Pradzny K., "The Interpretation of a Moving Retinal Image", Image Understanding 1984, Ablex Publishing Corp., 1984, pp. 179-193.

[5] LSI Logic Corporation, L64720 Motion Estimation Processor Data Sheets, pp. 5.

[6] Gilbert Strang, Linear Algebra and its Applications, Harcourt Brace Jovanovich, 1988, pp. 156

[7] Berthold Klaus Paul Horn, personal coversation during 4/96

[8] Anna Bruss and Berthold Klaus Paul Horn, "Passive Navigation", Computer Vision, Graphics, and Image Processing, 1983, pp. 18

[9] Jaochim Heel, "Direct Dynamic Motion Vision", Proceedings of the 1990 IEEE International Conference on Robotics and Automation, IEEE Computer Society Press, 1990, pp. 1142-1147

[10] Stephen William Lynn, personal conversation during 4/94

[11] ImageNation Corporation, Library Reference Manual for the CX100 & CX104, 1995, pp.16

[12] Berthold Klaus Paul Horn, personal coversation during 5/96

[13] Hennessy & Patterson, Computer Architecture a Quantitative Approach, Morgan Kaufman, 1996, pp. 29

[14] Berthold Klaus Paul Horn, personal coversation during 5/96