UNIVERSITY OF ALBERTA

Automatic Image Registration

BY

Collins Chien

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Science

DEPARTMENT OF COMPUTING SCIENCE

Edmonton, Alberta
Fall 1994

ISBN 0-315-95018-8

Name CELLINE CHIEN

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

COMPUTER SCIENCE — SUBJECT TERM

0984 — SUBJECT CODE

U·M·I

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS
Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION
General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523
Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS
Language
- General 0679
- Ancient 0289
- Linguistics 0290
- Modern 0291

Literature
- General 0401
- Classical 0294
- Comparative 0295
- Medieval 0297
- Modern 0298
- African 0316
- American 0591
- Asian 0305
- Canadian (English) 0352
- Canadian (French) 0355
- English 0593
- Germanic 0311
- Latin American 0312
- Middle Eastern 0315
- Romance 0313
- Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY
Philosophy 0422
Religion
- General 0318
- Biblical Studies 0321
- Clergy 0319
- History of 0320
- Philosophy of 0322

Theology 0469

SOCIAL SCIENCES
American Studies 0323
Anthropology
- Archaeology 0324
- Cultural 0326
- Physical 0327

Business Administration
- General 0310
- Accounting 0272
- Banking 0770
- Management 0454
- Marketing 0338

Canadian Studies 0385
Economics
- General 0501
- Agricultural 0503
- Commerce-Business 0505
- Finance 0508
- History 0509
- Labor 0510
- Theory 0511

Folklore 0358
Geography 0366
Gerontology 0351
History
- General 0578
- Ancient 0579
- Medieval 0581
- Modern 0582
- Black 0328
- African 0331
- Asia, Australia and Oceania 0332
- Canadian 0334
- European 0335
- Latin American 0336
- Middle Eastern 0333
- United States 0337

History of Science 0585
Law 0398
Political Science
- General 0615
- International Law and Relations 0616
- Public Administration 0617

Recreation 0814
Social Work 0452
Sociology
- General 0626
- Criminology and Penology 0627
- Demography 0938
- Ethnic and Racial Studies 0631
- Individual and Family Studies 0628
- Industrial and Labor Relations 0629
- Public and Social Welfare 0630
- Social Structure and Development 0700
- Theory and Methods 0344

Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES
Agriculture
- General 0473
- Agronomy 0285
- Animal Culture and Nutrition 0475
- Animal Pathology 0476
- Food Science and Technology 0359
- Forestry and Wildlife 0478
- Plant Culture 0479
- Plant Pathology 0480
- Plant Physiology 0817
- Range Management 0777
- Wood Technology 0746

Biology
- General 0306
- Anatomy 0287
- Biostatistics 0308
- Botany 0309
- Cell 0379
- Ecology 0329
- Entomology 0353
- Genetics 0369
- Limnology 0793
- Microbiology 0410
- Molecular 0307
- Neuroscience 0317
- Oceanography 0416
- Physiology 0433
- Radiation 0821
- Veterinary Science 0778
- Zoology 0472

Biophysics
- General 0786
- Medical 0760

EARTH SCIENCES
Biogeochemistry 0425
Geochemistry 0996
Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES
Environmental Sciences 0768
Health Sciences
- General 0566
- Audiology 0300
- Chemotherapy 0992
- Dentistry 0567
- Education 0350
- Hospital Management 0769
- Human Development 0758
- Immunology 0982
- Medicine and Surgery 0564
- Mental Health 0347
- Nursing 0569
- Nutrition 0570
- Obstetrics and Gynecology 0380
- Occupational Health and Therapy 0354
- Ophthalmology 0381
- Pathology 0571
- Pharmacology 0419
- Pharmacy 0572
- Physical Therapy 0382
- Public Health 0573
- Radiology 0574
- Recreation 0575
- Speech Pathology 0460
- Toxicology 0383

Home Economics 0386

PHYSICAL SCIENCES
Pure Sciences
Chemistry
- General 0485
- Agricultural 0749
- Analytical 0486
- Biochemistry 0487
- Inorganic 0488
- Nuclear 0738
- Organic 0490
- Pharmaceutical 0491
- Physical 0494
- Polymer 0495
- Radiation 0754

Mathematics 0405
Physics
- General 0605
- Acoustics 0986
- Astronomy and Astrophysics 0606
- Atmospheric Science 0608
- Atomic 0748
- Electronics and Electricity 0607
- Elementary Particles and High Energy 0798
- Fluid and Plasma 0759
- Molecular 0609
- Nuclear 0610
- Optics 0752
- Radiation 0756
- Solid State 0611

Statistics 0463

Applied Sciences
Applied Mechanics 0346
Computer Science 0984
Engineering
- General 0537
- Aerospace 0538
- Agricultural 0539
- Automotive 0540
- Biomedical 0541
- Chemical 0542
- Civil 0543
- Electronics and Electrical 0544
- Heat and Thermodynamics 0348
- Hydraulic 0545
- Industrial 0546
- Marine 0547
- Materials Science 0794
- Mechanical 0548
- Metallurgy 0743
- Mining 0551
- Nuclear 0552
- Packaging 0549
- Petroleum 0765
- Sanitary and Municipal 0554
- System Science 0790

Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY
General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: **Collins Chien**

TITLE OF THESIS: **Automatic Image Registration**

DEGREE: **Master of Science**

YEAR THIS DEGREE GRANTED: **1994**

(Signed) . . . . . .

Collins Chien
2002 G.H. Michener Park,
Edmonton, Alberta,
Canada T6H 5B5

Date: . . . . . .

Trust in the LORD with all thine heart;
and lean not unto thine own understanding.
In all thy ways acknowledge Him,
and He shall direct thy paths.
Be not wise in thine own eyes:
Fear the LORD, and depart from evil.
Proverbs 3:5–7 (KJV)

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Automatic Image Registration** submitted by **Collins Chien** in partial fulfillment of the requirements for the degree of **Master of Science**.

. . . . . .
Dr. T. A. Marsland (Supervisor)

. . . . . .
Dr. X. Li (Examiner)

. . . . . .
Dr. T. Spanos (External)

. . . . . .
Dr. W. W. Armstrong (Chair)

Date: . . . . . .

To my parents

Abstract

The purpose of image registration is to patch together several images to form a composite picture. Those images are captured by a camera positioned at different orientations and displacement. For example, a diver captures a series of underwater images of a ship's underside. Ship inspectors can use the technique of image registration to reconstruct the whole picture of the ship's underside without actually going underwater to inspect the seamless image of the ship.

Here an automatic image registration system was built with two major components: a position mapping module and a feature-based matching module. The mathematical model of position mapping is shown to be correct, given accurate positional information. Its low computational cost is competitive to any existing matching algorithm. Multiresolution feature-based matching is a versatile algorithm which can endure errors from the position mapping module and yet give an accurate match. It uses a multiresolution refinement technique to minimize the initial matching errors, and also converts from global to local search to reduce the computation. Experiments show that this system is working and has many practical applications.

Acknowledgements

I would like to thank my God who guide me through this work. He gives me life, wisdom and chances to explore new things. May all the glory be to my Heavenly Father, Lord Jesus Christ and the Holy Spirit.

Moreover, I really appreciate all the things that my parents and grandmother do to me in my life. Their unconditionally love, persistent encouragement and righteous guidance keep my head up when I am facing whatever problems. I owe them deeply that I can never repay. Of course, my two younger brothers who grow up with me, fight with me, play with me and love me. I thanks for their brotherly love and emotional support.

Margie, my girlfriend, is my major supporter since we met each other. Her love and support in all aspects have contributed a lot to my success.

I am very grateful that Professor Tony Marsland is my supervisor because he understand my difficulty. And each time after a meeting with him, I am encouraged and have energy to go on. My thanks also go to the examination committee, Dr. Spanos, Dr. Li and Dr. Armstrong. Dr. Li's advice led me to the correct direction. Dr. Armstrong's guidance during the teaching assistance with him gave me the best experience in teaching in my life so far.

Lastly, definitely not the least, I would like to thank my friends Dr. Kunyu Kwok, my big brother, who gives me advice and support whenever I need him. Gary Kwong, my prayer partner, who pray for me so that I can overcome all the difficulties and Dr. Ashraf Elnagar who contributes a lot in this thesis. And those who attend my presentation rehersal, Jimmy Kwong, Andrew Leung, Cheung Kwok Sing help me a lot in fixing my weaknesses.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Automatic image registration is a technique of putting together different (overlaying) images of the same scene, by either computing mapping functions between points or matching features in different images. This technique is useful in many real world situations such as medical imaging, satellite imaging, stereo vision and underwater inspection. In medicine [19], images of a patient's wound are registered together to detect changes of physical condition over a period of time. In aerospace [36], people propose to capture the aerial images of the planet Mars by attaching a camera to a balloon that is floating above the ground. And then by calculating the motion parameters of the floating balloon (i.e. translation and rotation) through registration techniques, one can determine the wind speed on the planet. Moreover, in underwater inspection [21], small fragment of ship images taken by divers are patched together to form a whole image of the ship. As a result, people can make better decisions about the maintainance plan. These examples il-

lustrate the usefulness of image registration technique in handling real world problems.

1.1 Definition of image registration

Image registration can be defined as a function which maps pixels from one image to another. Let

$\boldsymbol{p}_i \in \Re^2$ be the coordinates of a pixel in I_1 where $i = 1, \ldots, M$

$\boldsymbol{q}_j \in \Re^2$ be the coordinates of a pixel in I_2 where $j = 1, \ldots, N$

M and N are the total number of pixels in image I_1 and I_2 respectively. Then a continuous function f which maps $\boldsymbol{p}_i$ in I_1 to $\boldsymbol{q}_i$ in I_2 is

$$f \ : \ \Re^2 \rightarrow \Re^2 \text{ with } \boldsymbol{q}_i = f(\boldsymbol{p}_i).$$

Besides aligning spatial position between two images, intensity sometimes may also be matched as well.

$$g \ : \ \Re^2 \rightarrow \Re \text{ with } I_2(\boldsymbol{q}_i) = g(f(\boldsymbol{p}_i))$$

The goal of image registration is to find the optimal spatial and intensity transformation to determine the motion parameters or to expose the differences of interest between images.

1.2 Previous work

Several techniques were developed over years to handle different kinds of situations. Five major approaches and their merit and weakness are discussed

[4].

1.2.1 Correlation and Sequential

Correlation is a statistical approach for measuring the merit of a matching. It simply uses the intensity of the image as raw data for the comparison. It is mostly used in template matching where the image size is larger than the template. Equation (1.1) is called normalized cross-correlation [28]

$$C(u,v) = \frac{\Sigma_x \Sigma_y T(x,y) I(x+u, y+v)}{\sqrt{(\Sigma_x \Sigma_y I^2(x+u, y+v))}} \tag{1.1}$$

where $T(x,y)$ is the intensity value of the template and $I(x,y)$ is the image. Except for an intensity scale factor, if the template matches the image exactly, at a translation of (i,j) the cross-correlation will have its peak at $C(i,j)$. This correlation is directly derived from a more intuitive measure - the sum of difference square:

$$D(u,v) = \Sigma_x \Sigma_y (T(x,y) - I(x+u, y+v))^2 \tag{1.2}$$

When we expand Equation (1.2), we will get terms $\Sigma_x \Sigma_y T^2(x,y)$, $\Sigma_x \Sigma_y I^2(x+u, y+v)$ and $2\Sigma_x \Sigma_y T(x,y) I(x+u, y+v)$. Since the $T^2(x,y)$, square of the template values, is constant, we can drop it from the calculation. We then normalize the product term and obtain Equation (1.1).

By introducing μ_I and μ_T, the average intensity of the image and template respectively [29], we obtain Equation (1.3).

$$\frac{covariance(I,T)}{\sigma_I \sigma_T} = \frac{\Sigma_x \Sigma_y (T(x,y) - \mu_T)(I(x+u, y+v) - \mu_I)}{\sqrt{\Sigma_x \Sigma_y (I(x+u, y+v) - \mu_I)^2 \Sigma_x \Sigma_y (T(x,y) - \mu_T)^2}} \tag{1.3}$$

The range of this function is between -1 and 1. Under certain statistical assumptions, the value measured by the correlation coefficient gives a linear indication of the similarity between images. This is useful in order to quantitatively measure confidence or reliability in a match, and to limit the number of measurements needed to meet a prespecified confidence.

Equation (1.4) is a Fourier transform version of correlation.

$$\mathcal{F}(I(x,y) \circ T(x,y)) = \mathcal{I}(\omega_x, \omega_y) \times \mathcal{T}^*(\omega_x, \omega_y) \tag{1.4}$$

By the Correlation Theorem, the correlation of two functions $(I(x,y) \circ T(x,y))$ is equal to the multiplication of the Fourier transform of the functions $(\mathcal{I}(\omega_x, \omega_y) \times \mathcal{T}^*(\omega_x, \omega_y))$. Note that one of the Fourier transforms is in its conjugate form $(\mathcal{T}^*)$. This method is faster to compute than both equations (1.1) and (1.3), providing a rapid methods for calculating the Fourier transform.

Equation (1.5) is called sequential similarity detection algorithms [2].

$$E(u,v) = \Sigma_x \Sigma_y |T(x,y) - I(x+u, y+v)| \tag{1.5}$$

It accumulates the differences between the template and image and obtains the best match when the smallest value is calculated. It is called sequential because we can set a threshold value on the accumulated difference. Because we are doing the accumulation sequentially within a window, we can stop when the accumulation exceeds some threshold. If the sum of a single window does not exceed the threshold, we keep it and wait until all necessary windows are done and choose the smallest as indicating the best match. It is computationally simpler than all the above and terminates as soon as the accumulated value exceeds the threshold, thus saving time.

Equation (1.6) is a normalized version of Equation 1.5

$$E(u,v) = \Sigma_x \Sigma_y |T(x,y) - \hat{T} - I(x+u, y+v) + \hat{I}(u,v)| \qquad (1.6)$$

where $\hat{T}$ and $\hat{I}$ are the average intensity of the template and image respectively. It is useful because it gives an absolute measure of how the two images differ, regardless of their intensity scales.

Advantages and Disadvantages

These methods are generally useful for images that are misaligned by small rigid or affine transformations such as translation and rotation. They are intuitively easy to understand. However, it cannot handle complicated distortion for example, local distortion and images taken under a different environment. Moreover, as the type of transformation to be detected in the image increases, the computational cost and algorithm complexity increase dramatically.

1.2.2 Fourier Methods

Kuglin and Hines [22] proposed an elegant phase correlation method, which aligns two images that are shifted relative to one another. They used the shift property of Fourier transform:

$$\mathcal{F}[f_2(x,y)] = \mathcal{F}[f_1(x-d_x, y-d_y)] = F_2(\omega_x, \omega_y) = e^{j(\omega_x d_x + \omega_y d_y)} F_1(\omega_x, \omega_y) \qquad (1.7)$$

where two images f_1 and f_2 which differ only by a displacement (d_x, d_y). We can obtain (d_x, d_y) by computing the cross-power spectrum of the two

images:

$$\frac{F_1(\omega_x,\omega_y)F_2^*(\omega_x,\omega_y)}{|F_1(\omega_x,\omega_y)F_2^*(\omega_x,\omega_y)|} = e^{(\omega_x d_x+\omega_y d_y)} \tag{1.8}$$

where F^* is the conjugate of F. In order to search for the rotation transformation, we can change the coordinate system to polar [7] in which Equation (1.8) can be written in terms of the rotating angle.

Advantages and Disadvantages

This method gives more information, such as the correlation, than just a similarity measure. It actually provides how far the template should move in order to fit with the image. Moreover, it can tolerate some scene noise such as brightness and shade because those effect will appear at the low frequency region in frequency domain. However, the result of this method may be inaccurate when random noise is dominating the image. Since random noise exists in the whole range of frequency domain, it corrupts the phase difference. Moreover, this method does not handle local distortion.

1.2.3 Control Points

Explicit control points are specially placed in the scene to aid registration. The matching is done by a human, or by a machine when amount of data is overwhelmingly large. It is usually done by cross-correlation since high accuracy is desired at this level and since the template size is small enough so the computation is feasible.

Implicit control points are determined from data, either manually or automatically. Typical automatically found control points are corners, lines

and intersections. Matches can be determined based on the properties of these points such as curvature or the direction of the principal axes. Other techniques such as clustering and relaxation are used to match and determine optimal transformation.

Point Mapping with Feedback

Using feedback between the stages of finding control point matches and finding the optimal transform, these methods can successfully register images where automatically acquired features are ambiguous or when there are significant amounts of uncorrected local variations. Techniques used usually involved iteration or hierarchical processing structure such as relaxation [31] and multiresolution [36].

Point Mapping without Feedback - Global point matching

Non-feedback point mapping methods [17, 24] are used to register images for which the transformation necessary to align the images is unknown. As a result, a global function is used to fit to the unknown transformation. Two approaches can be taken,

1. Approximation: parameters of transformation are found so that points matched as nearly as possible. This approach uses statistical methods and many control points for accurate approximation.

2. Interpolation: fewer control points are needed but they must be accurate for example good intrinsic and manual control points. There

must be precisely one matched point for each independent parameter of the transformation to solve the system of equations. But when control points increase, use of a high order of polynomial will produce unexpected undulations. As a result, least square or splines and other piecewise interpolations are preferable.

Advantages and Disadvantages

Point matching greatly reduces the size of information to be considered relative to correlation and Fourier transform. Either matching with or without feedback can tolerate global and local uncorrected variation because of the use of control points and local similarity measures. They use information from spatial relationships between control points in the image and they can consider possible matches based only on supporting evidence. However, for feedback matching, a more sophisticated search algorithm is needed because the local uncorrected variations make the search space more difficult to traverse. Moreover, these techniques are limited to small affine transformations because otherwise the search space becomes too large and unmanageable. The major drawback of global point matching method is that it cannot account for local geometric distortion.

1.2.4 Local Method - Piecewise Interpolation

Only control points sufficiently close, or perhaps, weighted by their proximity to the interest region, influence each part of the mapping transformation [13]. The parameters of the local mapping transformation vary across the

different regions of the image. It can handle many distortions that global methods cannot, for example complex 3D scenes taken from different viewpoints, deformable objects or motions, and the effects of different sensors or scene conditions.

Presumption: control points are found and matched before the determination of the registration transformation.

Advantages and Disadvantages

Since a piecewise interpolation method use more complex calculations, it is computationally intensive and slower than a global method. Moreover, because a local method does not carry feedback information in the calculation, it depends heavily on the accuracy of the similarity measure in this single-pass process. However, because of this single-pass property, a local method does not need complicated algorithm to cycle through multi-passes of iteration like the feedback algorithms.

1.2.5 Elastic Model-Based Matching

Elastic model-based matching methods [1, 5, 30] model the distortion in the image as the deformation of an elastic material. In other words, the registration transformation is the result of the deformation of an elastic material with the minimal amount of bending and stretching. The amount of bending and stretching is characterized by the energy state of the elastic material.

Three aspects of this method:

1. Iteration: at each iteration step features are found, and a correspon-

dence measure is determined which influences a transformation which is then performed before the sequence is repeated.

2. Hierarchical structure: larger and more global distortions are corrected first. Then progressively smaller and more local distortions are corrected until a correspondence is found which is as finely matched as desired.

3. Cooperation: features in one location influence decisions at others.

Advantages and Disadvantages

Elastic model-based matching has the advantage that it corrects local distortion. It simply rectifies the distortion with its elastic sheet property. However, objects in the image must be predefined. This may be a difficult problem because most of the real objects are hard to be modeled.

1.3 Problem Statement

In underwater inspection, a diver is video-taping the underwater scene of a structure to determine if repair is needed. It is always difficult to imagine the condition of a huge structure just by looking at a small part of it. As a result, the inspector wants to patch images together from the divers to view the whole structure. Image registration techniques are needed to accomplish the task. A set of assumptions is needed to limit the scope of the problem being solved.

1. *Position information is available*

   Given position and orientation data such as pan and tilt angle of the camera, one can use geometrical (or global projective) transformation to register two images together. If these data are not available, a pure feature-based registration must be used and the computational cost will be extremely high.

2. *Camera's properties are known*

   The geometrical model needs the camera's properties, such as focal length, for handling translation movement along x, y and z direction.

3. *Images have arbitrary and unknown distortion such as perspective and atmospherical distortion*

   Perspective viewing [18] and back scattering [21] are natural phenomena that can be compensated by global projective transformation [32] and image enhancement.

4. *Objects and background in the images are stationary (no motion blur and no growth with time)*

   Occlusion and object movement are not recoverable by geometrical transformation. Objects being viewed should be stationary and the registration session should be done within a short time.

1.3.1 Related Problems

1. Position information mapping

   It is important to have a mathematical model using position and ori-

entation information to map points in space. If this mapping function gives closely registered images, the processing time of the feature-based refinement will be greatly reduced.

2. Feature extraction

   When inaccurate position information is given and local distortion results from perspective viewing, I use the information in the image to correct the error. A feature extraction algorithm is needed to obtain a useful feature point to do the final refinement matching.

3. Feature points correspondence

   Once the feature points are available, one must find the correspondence of feature points between two images so that proper correction can be made.

1.4 System Overview

An automatic image registration system (see Figure 1.1) is built to solve the problem. Since the movement and orientation of the camera is known, the image distortion caused by perspective viewing can be rectified by a set of geometrical transformation functions. The details and derivation of these functions are described in Chapter 2. Once the rectified information is available, images are warped according to the new shape and form a preliminary matching.

Although the mathematical model provides the exact position and image transform to the patching, it is highly susceptible to inaccurate position

data. I classify this inaccuracy as error from the position information system. The major error occurs when the physical position and orientation do not correspond to the data reported to the registration system. As a result, incorrect transformation gives undesirable matching. To rectify this error, I choose a feature-based matching (FBM) technique which uses the information contained in the image. FBM, as I have mentioned in Section 1.2.3 - point-mapping with feedback - can tolerate local distortion and is a fully automatic process. Through feedback at each stage of finding control point correspondence, a accurate matching results. FBM extracts feature points from the warped images (Chapter 3) and then finds the point correspondence (Chapter 4) between images. Once a correspondence relationship is established, a proper affine transformation can correct the error caused by the positioning system.

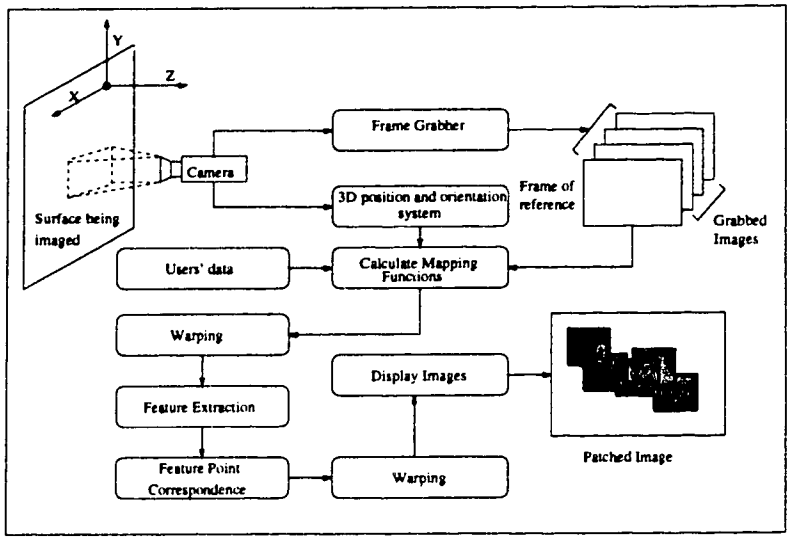


Figure 1.1: *Image Registration System Diagram*

Chapter 2

Position mapping

2.1 Mathematical model

Here we develop a theory that determines the position of corresponding pixels in two images of the same scene. We assume that information about the position of the camera is available. Next we define the coordinate systems used and derive relationships between images.

Throughout this work, the pinhole camera model is used. As Figure 2.2 shows, the origin of a cartesian coordinate system $OXYZ$ is the viewpoint of the camera and its optical axis is aligned with the Z axis. The image is the projection of a 3D scene onto a plane located at distance f (focal length) from O. The image coordinate system is represented by oxy in Figure 2.2. Using the above model and assuming perspective projection, the relationships between points in the image plane and points in 3D with respect to the camera coordinate system are:

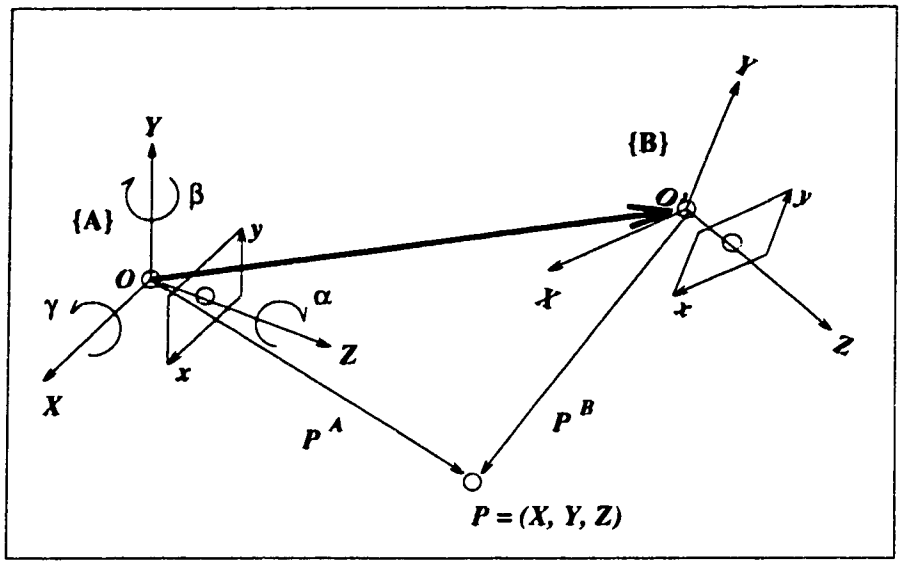


Figure 2.2: *Different coordinate systems.*

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

where $P = (X, Y, Z)$ is a point in the camera coordinate system, and $p = (x, y)$ is its projection on the image plane; f_x is the number of pixels corresponding to the focal length (f) along the x-axis; f_y is the number of pixels corresponding to f along the y-axis.

To find the correspondence mapping of a point in two different frames, choose $P^B = (X^B, Y^B, Z^B)$ to be an arbitrary 3D point in frame $\{B\}$. $P^A = (X^A, Y^A, Z^A)$ (P with respect to $\{A\}$) is obtained by the following relation

(Figure 2.2):

$$\begin{bmatrix} (P^A)^{\mathbf{T}} \\ 1 \end{bmatrix} = T_B^A \begin{bmatrix} (P^B)^{\mathbf{T}} \\ 1 \end{bmatrix} \tag{2.1}$$

where T_B^A is a 4×4 homogeneous transformation matrix defined as :

$$T_B^A = \begin{bmatrix} r_{11} & r_{12} & r_{13} & \delta_x \\ r_{21} & r_{22} & r_{23} & \delta_y \\ r_{31} & r_{32} & r_{33} & \delta_z \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

The 3×3 sub-matrix ($R_B^A = [r_{11}...r_{33}]$) represents the rotation matrix relating $\{B\}$ to $\{A\}$, and $[\delta_x, \delta_y, \delta_z]$ denotes the displacement vector from the origin of $\{A\}$ with respect to $\{B\}$ (OO' in Figure 2.2). We use the Z-Y-X Euler angles approach to find the rotational matrix, R_B^A:

$$R_B^A(\alpha, \beta, \gamma) = ROT(Z, \alpha)ROT(Y, \beta)ROT(X, \gamma)$$

where α, β, and γ are the angles rotated about the Z, Y, and X axes respectively. This means that we start with a frame that is coincident with $\{B\}$ and rotate this frame about Z by an angle α, then rotate the resulting frame about Y by an angle β, and finally rotate about X by an angle γ. As a result, the rotation matrix takes the form:

$$R_B^A(\alpha, \beta, \gamma) = \begin{bmatrix} c\alpha & -s\alpha & 0 \\ s\alpha & c\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c\beta & 0 & s\beta \\ 0 & 1 & 0 \\ -s\beta & 0 & c\beta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & c\gamma & -s\gamma \\ 0 & s\gamma & c\gamma \end{bmatrix}$$

$$= \begin{bmatrix} c\alpha c\beta & c\alpha s\beta s\gamma - s\alpha c\gamma & c\alpha s\beta c\gamma + s\alpha s\gamma \\ s\alpha c\beta & s\alpha s\beta s\gamma + c\alpha c\gamma & s\alpha s\beta c\gamma - c\alpha s\gamma \\ -s\beta & c\beta s\gamma & c\beta c\gamma \end{bmatrix}$$

where $c\theta \equiv \cos\theta$ and $s\theta \equiv \sin\theta$. Substituting in Equation (2.1) we obtain:

$$X^A = (c\alpha c\beta)X^B + (c\alpha s\beta s\gamma - s\alpha c\gamma)Y^B + (c\alpha s\beta c\gamma + s\alpha s\gamma)Z^B + \delta_x \tag{2.2}$$

$$Y^A = (s\alpha c\beta)X^B + (s\alpha s\beta s\gamma + c\alpha c\gamma)Y^B + (s\alpha s\beta c\gamma - c\alpha s\gamma)Z^B + \delta_y \tag{2.3}$$

$$Z^A = (-s\beta)X^B + (c\beta s\gamma)Y^B + (c\beta c\gamma)Z^B + \delta_z \tag{2.4}$$

Recall that perspective projection equations in frames $\{A\}$ and $\{B\}$ are:

$$x^A = f_x\frac{X^A}{Z^A}, \qquad y^A = f_y\frac{Y^A}{Z^A}$$
$$x^B = f_x\frac{X^B}{Z^B}, \qquad y^B = f_y\frac{Y^B}{Z^B}$$

Now substituting Equations (2.2, 2.3 and 2.4) in the above relations yield:

$$x^A = \frac{x^B(c\alpha c\beta) + y^B\frac{f_x}{f_y}(c\alpha s\beta s\gamma - s\alpha c\gamma) + f_x(c\alpha s\beta c\gamma + s\alpha s\gamma) + f_x\frac{\delta_z}{Z^B}}{-\frac{s\beta}{f_x}x^B + \frac{c\beta s\gamma}{f_y}y^B + c\beta c\gamma + \frac{\delta_z}{Z^B}} \tag{2.5}$$

$$y^A = \frac{x^B\frac{f_y}{f_x}(s\alpha c\beta) + y^B(s\alpha s\beta s\gamma + c\alpha c\gamma) + f_y(s\alpha s\beta c\gamma - c\alpha s\gamma) + f_y\frac{\delta_y}{Z^B}}{-\frac{s\beta}{f_x}x^B + \frac{c\beta s\gamma}{f_y}y^B + c\beta c\gamma + \frac{\delta_z}{Z^B}} \tag{2.6}$$

Notice that, aside from the last term in the numerator and the denominator, these equations are dependent on image plane information only.

2.1.1 Rotational case

If we assume rotation only (i.e., $\delta_x = \delta_y = \delta_z = 0$), correspondence between pixels in both images is directly established (Equations 2.5 and 2.6 without the last terms in both the numerators and the denominators). If we further assume that no rotation about the Z-axis is allowed (i.e., $\alpha = 0$) then we obtain the equations of a pan-tilt system only:

$$\begin{aligned} x^A &= \frac{x^B c\beta + y^B \frac{f_x}{f_y}(s\beta s\gamma) + f_x(s\beta c\gamma)}{-\frac{s\beta}{f_x}x^B + \frac{c\beta s\gamma}{f_y}y^B + c\beta c\gamma} \\ y^A &= \frac{y^B c\gamma - f_y s\gamma}{-\frac{s\beta}{f_x}x^B + \frac{c\beta s\gamma}{f_y}y^B + c\beta c\gamma} \end{aligned}$$

If the system supports pan-only rotation, then:

$$\begin{aligned} x^A &= \frac{x^B c\beta + f_x s\beta}{-\frac{s\beta}{f_x}x^B + c\beta} \\ y^A &= \frac{y^B}{-\frac{s\beta}{f_x}x^B + c\beta} \end{aligned}$$

Similarly if the system allows tilt-only rotation, then:

$$\begin{aligned} x^A &= \frac{x^B}{\frac{s\gamma}{f_y}y^B + c\gamma} \\ y^A &= \frac{y^B c\gamma - f_y s\gamma}{\frac{s\gamma}{f_y}y^B + c\gamma} \end{aligned}$$

2.1.2 Translational case

It is clear from Equations (2.5) and (2.6) that the depth at every point in the image is necessary to recover the effect of translational motion in the registration process for precise correspondence between image points in both

images. However, this process is not only complex but also time consuming unless we also use a range scanner positioning system to provide such information. If translation motion between images is minimal and the surface (plane) on which images are registered is far from the camera, we can scale the new image (after motion) and accordingly compute its four corners with respect to the frame of reference of the original image. As a result, the procedure to registering two images consists of two steps. The first step treats the orientation (Section 2.1.1) and the second one deals with the translation. Assuming no translation, Equations (2.5) and (2.6), are sufficient to recover the rotational effect. Having the same orientation, for both images to be registered, we scale the new image according to the translational motion that took place between both frames. Formally, given a point $p^B = (x^B, y^B)$ that belongs to the new image, its corresponding point in the original frame of reference (say $p^A = (x^A, y^A)$) after recovering orientation is determined by:

$$\begin{aligned} x^A &= x^B(1 - \frac{\delta_z}{Z}) + \delta_x \\ y^A &= y^B(1 - \frac{\delta_z}{Z}) + \delta_y \end{aligned}$$

where Z is the depth to the surface (plane) on which registration is performed.

This solution is an approximate one since depth to points in a general scene usually varies from one to another. However, in some specific applications, such as viewing a 2D surface, bathymetry (i.e., sea floor), or registration of air images (i.e., aerial view of an urban area), we can approximately assume the scene as a surface, especially if Z is large enough.

With the availability of the current technology, we can use a ranging device to recover the depth at each point in the scene. As a result, the

effect of translational motion can be easily determined from equations (2.5) and (2.6). Assuming the camera is translating only (no rotation), a point $p^B = (x^B, y^B)$ is mapped to $p^A = (x^A, y^A)$ by the following relations:

$$x^A = \frac{Z^B x^B + f_x \delta_x}{Z^B + \delta_z}$$
$$y^A = \frac{Z^B y^B + f_y \delta_y}{Z^B + \delta_z}$$

Henceforth, Equations (2.5) and (2.6) can be used to register two (or more) images that are offset by rotation and/or translation. The process of registering multiple images can be easily done by either treating each new image with respect to the previous one, or with respect to the original frame of reference (first image in the sequence).

2.2 Warping algorithm

Since our mathematical model can handle all kinds of motion under perspective viewing, it can compensate any perspective distortion due to the rotation of the camera. For example, if the camera pans, to the right, the distortion compensated image will be a trapezoid having the shortened edge on the left and the elongated edge on the right (see Figure 2.3).

For this reason we need to warp a regular image into trapezoid shape in order to compensate the distortion. Figure 2.4 shows the effect with and without warping. Moreover, the mathematical model that we introduced is simply a set of point mapping functions. Theoretically, we should use these equations to map every pixel in one image onto the frame of reference image. However, this has high computational cost and suffers from the "hole effect".

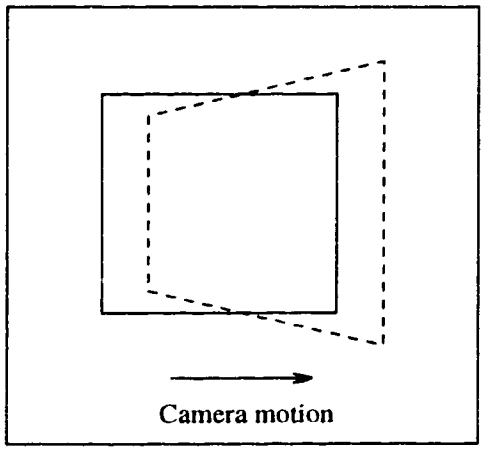
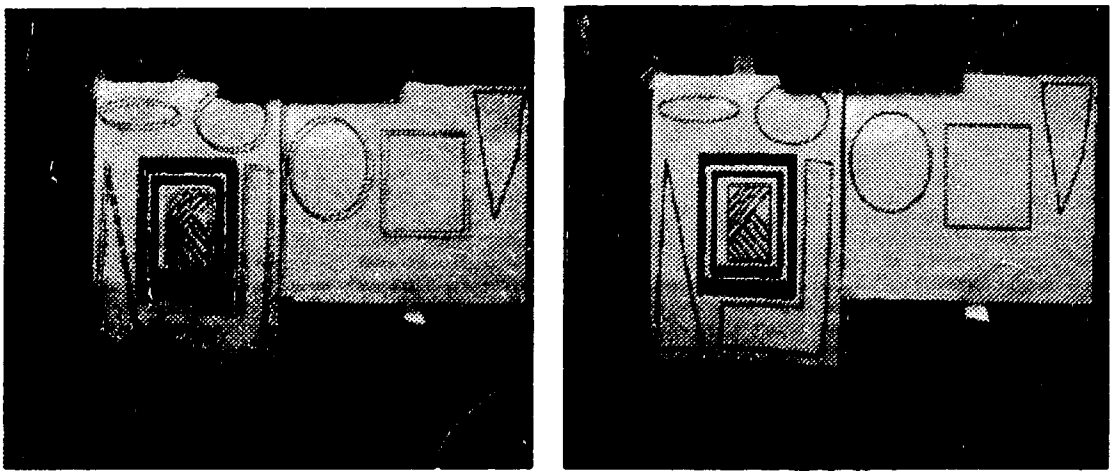


Figure 2.3: *Perspective Distortion in panning*



(a) *(b)*

Figure 2.4: *(a) Without warping and (b) with warping*

The "hole effect" occurs because the x and y coordinate values obtained from the mapping functions are decimal numbers. As a result, some of the pixels may be mapped onto the same position due to round off. Our solution is to simply map the four corners of the image onto the frame of reference and then use a warping technique to fill in the pixels.

Image warping (or geometrical transformation) is a process that maps the spatial position and interpolates grey level values of a source image to a destination image. There are two main approaches: forward and backward mapping. By forward mapping, the input of the mapping function is the source image pixel and the output is the location of that pixel in the destination image coordinate system. On the other hand, backward mapping is just the reverse process. We used backward mapping because we can obtain grey level values easily by the bilinear method.

The warping function is defined by a pair of equations of the form

$$x = f(u, v), \qquad y = g(u, v)$$

where u and v are the coordinates of a pixel on the destination image plane and x and y are the calculated position on the source plane. In our implementation, f and g are polynomial functions.

$$x = f(u, v) = \sum_{l=0}^{n} \sum_{k=0}^{n} a_{lk} u^l v^k$$

$$y = g(u, v) = \sum_{l=0}^{n} \sum_{k=0}^{n} b_{lk} u^l v^k$$

where a_{lk} and b_{lk} are the coefficients of polynomials f and g respectively, and n is the degree of both polynomials. Usually, the degree of a polynomial is

determined by the number of control points available. However, because of the well-known unstable effect caused by high-degree polynomial, one of at most 3rd degree is used. Since we use the four corners of the image plane as control points, the maximum degree of the polynomial is one.

$$x = a_{00} + a_{10}u + a_{01}v + a_{11}uv \tag{2.7}$$

$$y = b_{00} + b_{10}u + b_{01}v + b_{11}uv \tag{2.8}$$

and we can write it in the form of matrix

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 & u_0 & v_0 & u_0v_0 \\ 1 & u_1 & v_1 & u_1v_1 \\ 1 & u_2 & v_2 & u_2v_2 \\ 1 & u_3 & v_3 & u_3v_3 \end{bmatrix} \begin{bmatrix} a_{00} \\ a_{10} \\ a_{01} \\ a_{11} \end{bmatrix}$$

To find the a's, we take the inverse of the square matrix (M) and multiply it to both sides

$$\vec{a} = M^{-1}\vec{x}$$

Once we find the polynomial, we can locate the corresponding position of a destination pixel in the source plane. We use a bilinear (or first degree) interpolation method to obtain the grey level value from the source plane.

2.3 Registration algorithm

Our implementation of the position mapping system is shown as a block diagram in Figure 2.5 (It is the same as Figure 1.1 but for reader's convenience, I redisplay it here). Initially, a scene is captured by the computer

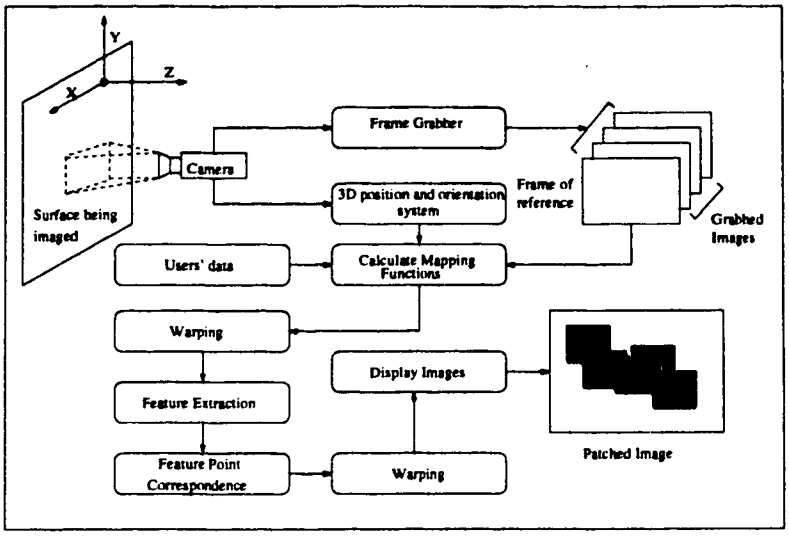


Figure 2.5: *Main components of image registration system (repeat Figure 1.1).*

controlled camera and the image is digitized by the image grabber. The first captured image is the frame of reference image and the images after it are mapped with respect to it. Once user's data, such as focal length of the camera lens and position and orientation information are available, the coordinates of the four corners of the image after certain motion are computed using the mapping functions introduced in Section 2.1. Since only four points are needed to calculate the mapping, this algorithm is computationally efficient. Moreover, it is grey level insensitive because only positional information is used in the calculation. As we have explained in a previous section, all the pixels inside the newly shaped image are filled by the warping algorithm. Then the warped images are integrated and displayed on the screen.

2.4 Experimental Results

We have conducted several experiments to test our mathematical model. In order to obtain accurate position and orientation information, we use a computer to control the COHU video camera that is mounted on a pan/tilt device (see Figure 2.6). The input values of the pan/tilt device is "points"

Figure 2.6: *The COHU camera with its pan/tilt mounting*

which is defined by the manufacturer. With the same input points, the pan movement is different from the tilt. For example, 20 points translates into 1.67° of pan movement. On the other hand, 20 points of tilt means 0.99° of tilting. Table 2.1 is the conversion that is used in our experiment. Moreover, since the COHU camera unit is fixed on a table, we must translate the object in order to simulate the camera translation movement. To do that, we slide the object on a horizontal plane to simulate the x and z translation. For the y translation, we lift the object with a $1.8cm$ thick plastic spacer.

Points	Pan (degree)	Tilt (degree)
20	1.67	0.99
30	2.51	1.49
50	4.18	2.48
100	8.37	4.96

Table 2.1: *Point to pan/tilt movement conversion table*

2.4.1 Pan and tilt

First of all, we pan the camera to the right at 20 points intervals for 6 frames (Figure 2.7 (a) to (f)). One should notice that the final patched image (Figure 2.7 (g)) is a trapezoid as explained in Section 2.2. This kind of effect can be seen in daily life when we project a beam of light that is not perpendicular to the wall. Secondly, we tilt upward for 30 points steps for 6 frames (Figure 2.8 (a) to (f)). The same trapezoidal effect appears in the patched image result (Figure 2.8 (g)) but in a different direction. Finally we combine both pan and tilt in order to cover the whole board (Figure 2.9). The frame of reference is at the center, so the trapezoidal effect is equal in all four directions.

These examples show that our model can accurately register two or more images together. We do not use a test for rotation about the viewing axis (z axis), because the set up of our camera does not allow us to do this kind of rotation without affecting other parameters, such as horizontal and vertical position. However, our mathematical models takes it into account.

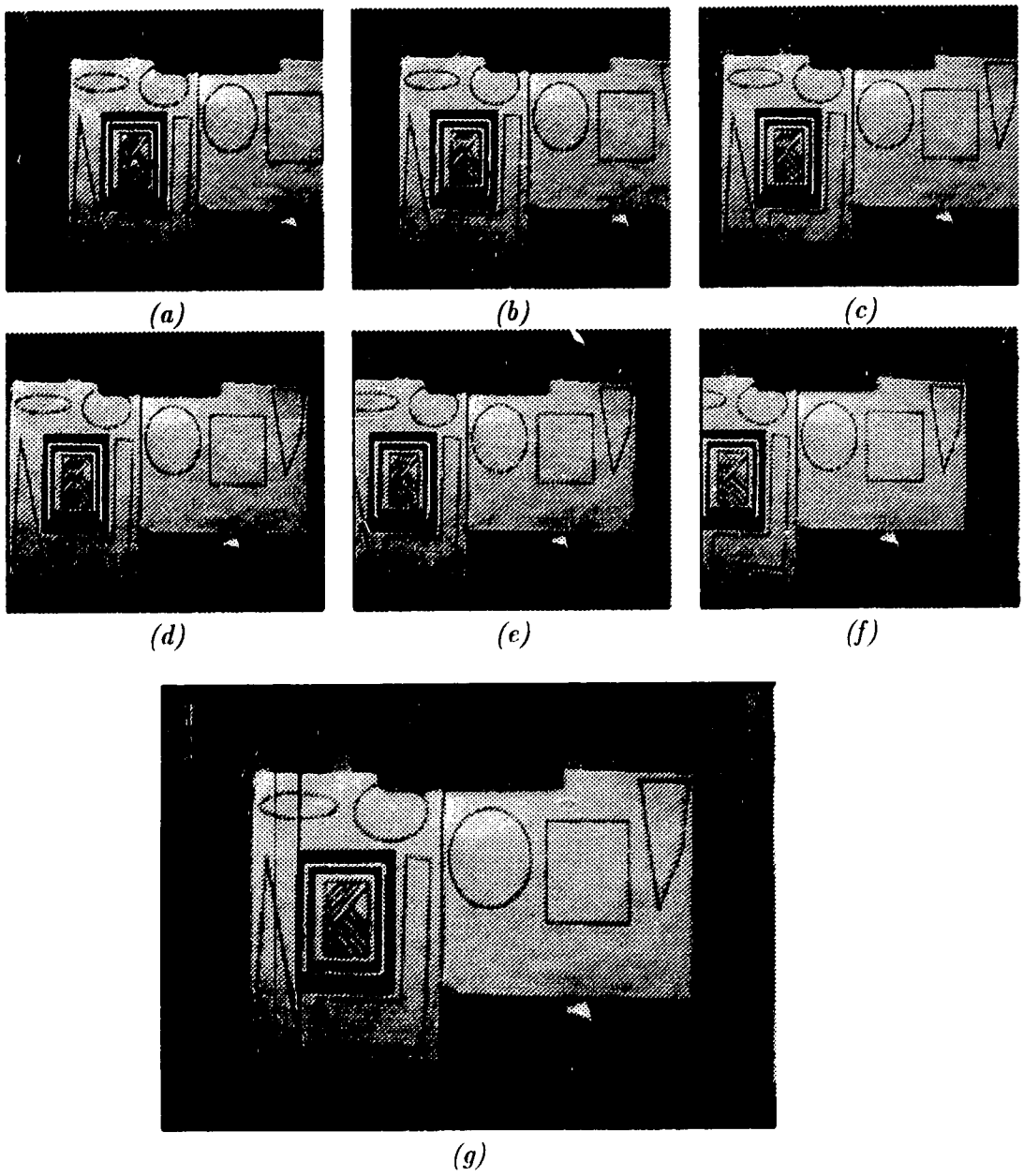


Figure 2.7: *Image sequence of panning (a) to (f) and the patched image (g)*

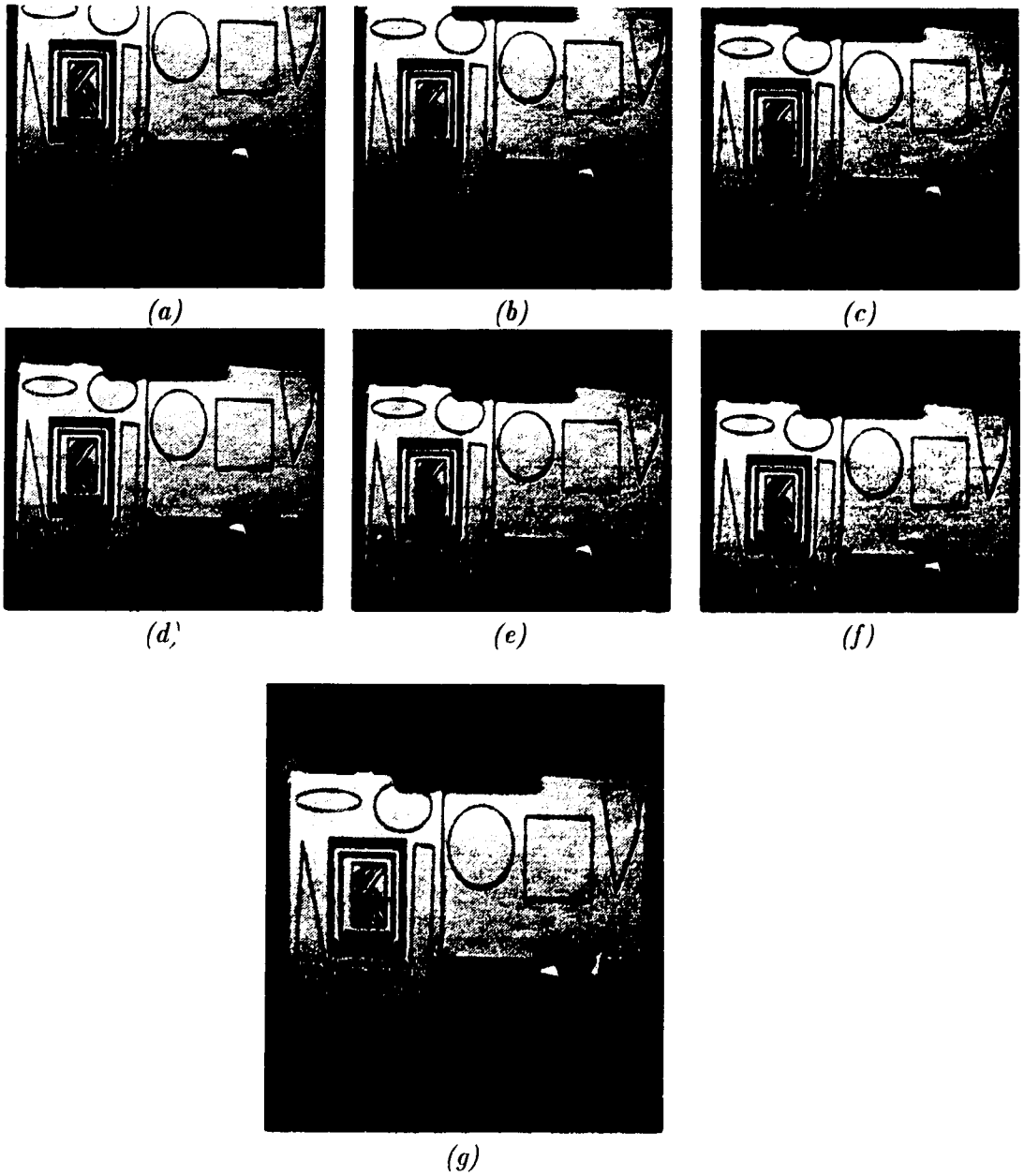


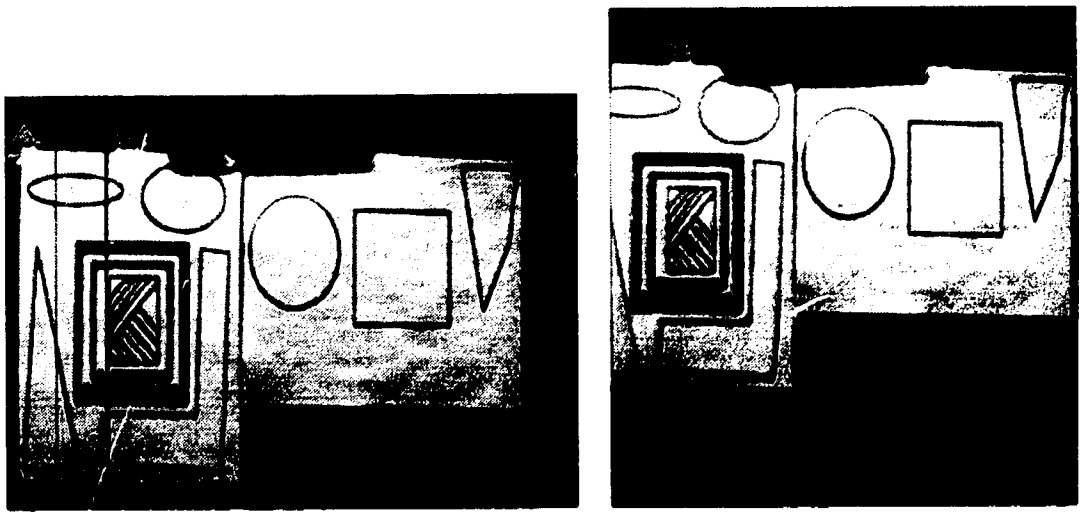
Figure 2.8: *Image sequence of tilting (a) to (f) and the patched image (g)*

Figure 2.9: *Pan and tilt combination*

2.4.2 Translation

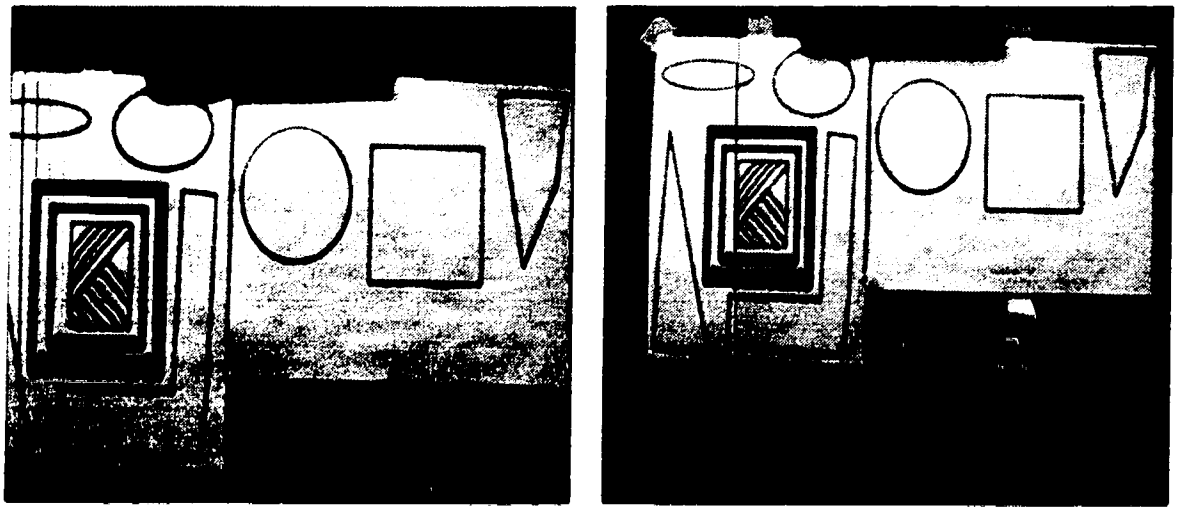
In the following three experiments, we test our model for translational motion. Normally we would translate the camera, but because of our set up we move the board instead. First, we shift the board from left to right at 5 centimeter interval for three frames and obtain Figure 2.10 (a). In the second experiment, the board is moved upward with 1.8 centimeter interval for five frames with the resulting image in Figure 2.10 (b). And in Figure 2.11 (a), we move the board backward from its initial 75 centimeter distance from the camera at five centimeter intervals for five frames. Since they are only translations, the images are in the original shape without any perspective distortion.

In Figure 2.11 (b) we present results from different experiment, in which



(a) *(b)*

Figure 2.10: *Horizontal (a) and vertical (b) translations*



(a) *(b)*

Figure 2.11: *Longitudinal (a) and combined (b) translations*

we combine all three translations with 25 centimeter longitudinally backward, 5 centimeter leftward and 5.4 centimeter (i.e. 3 spacers) downward.

2.4.3 Pan/tilt and translation combination

We now combine every possible motion (except rotation about z axis) to show the robustness of our model. The image sequence is arranged in the following order:

1. Frame of reference image (Figure 2.12 (a))
2. Longitudinally translation for 10 centimeter (Figure 2.12 (b))
3. Pan left for 20 points (Figure 2.12 (c))
4. Vertically displacement downward for 5.4 centimeter (Figure 2.12 (d))
5. Tilt downward for 50 points (Figure 2.12 (e))
6. Horizontally movement rightward for 5 centimeter (Figure 2.12 (f))

2.4.4 A Real scene

We have also tested our model with a realistic scene. By realistic we mean that objects in the image are not just a flat wall but an arbitrary indoor scene. Again, we can only test the pan and tilt since our camera is fixed. The sequence of images are shown in Figure 2.13 where *(a)* to *(d)* are panning from left to right for 50 points interval and *(e)* to *(h)* are panning from right to left at the same intervals after a tilt of 100 points.

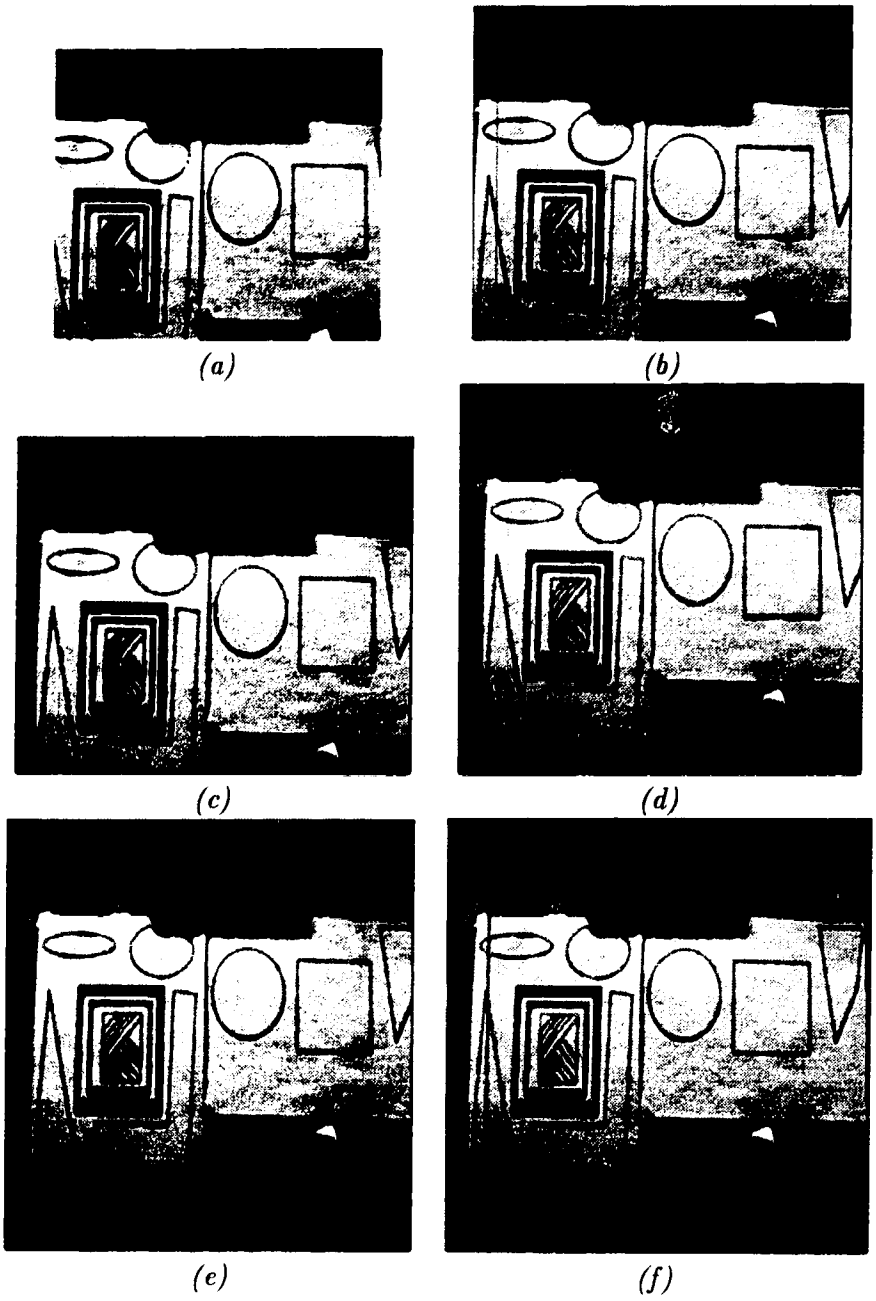


Figure 2.12: *Combination of pan/tilt and translation*

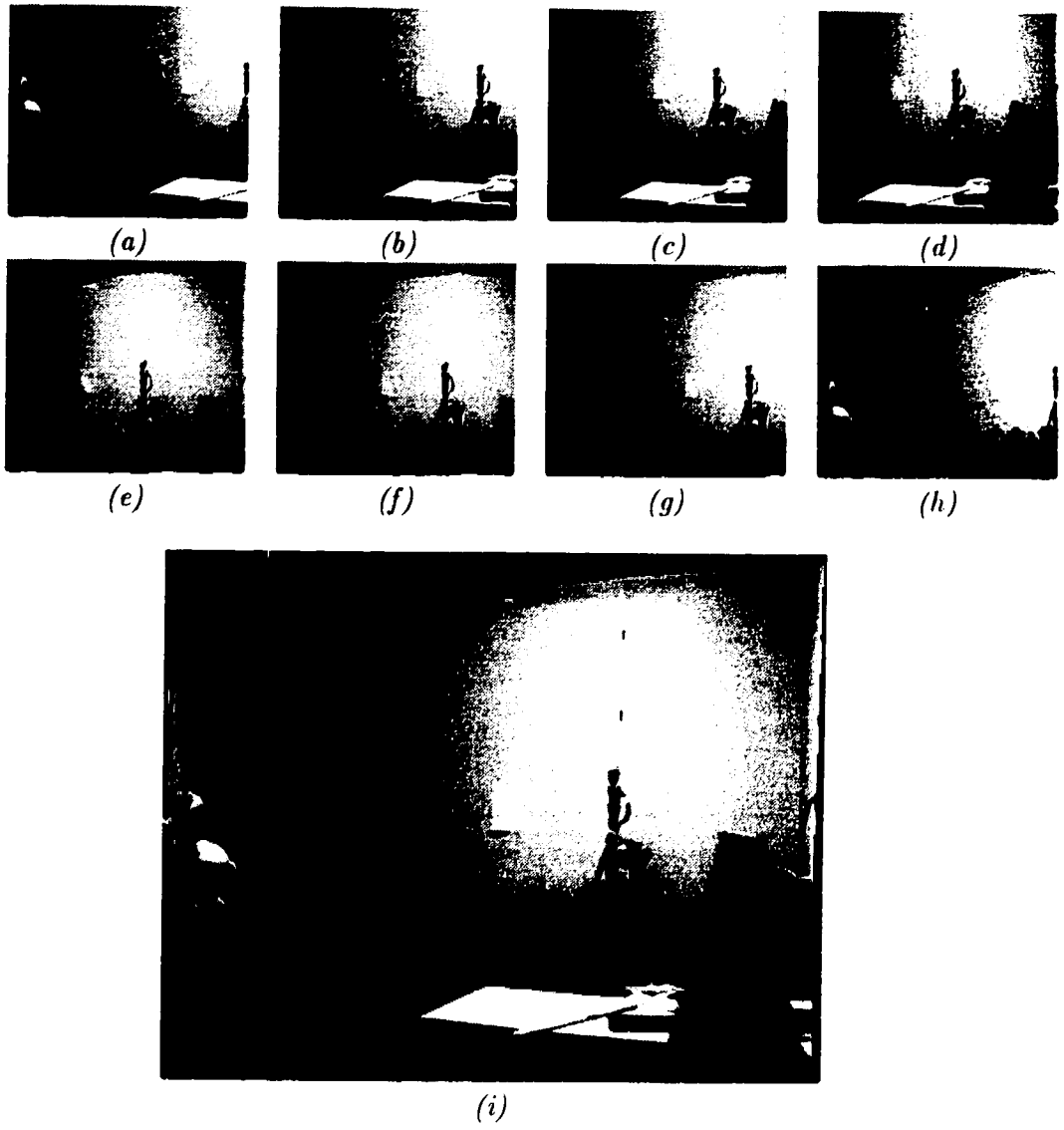


Figure 2.13: *Image sequence of real scene (a) to (h) and resulting patched image (i)*

Chapter 3

Feature Detector

The notion of a feature has a large range of definition such as points, lines, corners, shapes, shadow, colors and even semantic properties. When explicit control points are used, the landmarks in the image are the feature points. For example, Cheng and Huang [9] used edges of a car to form a construct and to match with the shape of the car from other images.

Among these vast choices of feature, one must decide a feature to extract so that a correspondence can be found. First of all, the feature must be distinctive (i.e. be different from neighboring points). For example, points on edges should not be selected because they look the same along edges. Secondly, the feature should be invariant with respect to geometrical distortions. If large geometrical distortion is expected, a specific shape is not a good choice for feature because the basic shape may not be retained. A corner satisfies both distinctiveness and invariance requirements.

By definition, a point is a corner when a vector changes its direction at

that point. A corner is distinct enough to be separated from its neighbor. Moreover, in our application of underwater inspection, the expected geometrical distortion is small enough that corner can be classified as invariant to the distortion.

Many corner detection methods have been developed over the years. Seven methods are selected for comparison in this chapter. They are chosen because they represent the two major approaches of corner detection: surface fitting [3, 11, 20, 34, 38], template matching or interest operator [8, 10, 15, 16, 23, 27, 37]. One newer technique transforms corner detection into a cost minimization problem [35]. However, its computational cost is too high so that I did not implement it. Each approach has its advantages and disadvantages so each is evaluated based on the following criteria.

1. *Stability*

   The selected points should be expected to appear in the other images. Thus the selection should be robust with respect to noise.

2. *Interpretability*

   The method should detect all the well-defined corners that exist in the image. A well-defined corner is one that a normal-sighted human will also label that feature as a corner.

3. *Speed*

   The method should finish the detection process in reasonable time because the system is expected to be working in real time.

3.1 Surface Fitting Techniques

Suppose a scene is a continuous two dimensional function of grey level value $S(x, y)$. A digital image is a discrete version of this function $S_d(u, v)$ where u and v are integer values. The frequency of sampling depends on the density of those camera optical sensors. In other words, each pixel in the image is a sample of that continuous function. We can use either a high resolution camera or surface fitting to approximate $S(x, y)$. High resolution means that we are sampling $S(x, y)$ at a frequency that is high enough to approximate $S(x, y)$ accurately. However, the high price of a high resolution image digitizer prohibits further development on this approach. Because of this constraint, a surface fitting approach is used. The basic idea is to fit a continuous function over a small processing region. This processing region is defined by the size of neighborhood surrounding the central pixel. Then a method such as least squares fit is used to find all the coefficients of the surface. Once the equation of a continuous surface $S'(x, y)$ is found, we can calculate the gradient ($\nabla S'(x, y)$), Laplacian ($\nabla^2 S'(x, y)$) and the curvature of the central pixel of that region. If the curvature is greater than certain threshold value, a corner is found.

3.1.1 Turning of fitted surface

Kitchen and Rosenfeld [20] used the approximate method to find the point curvature. They assumed that a surface $g(x, y)$ fits the gray level of the image. The gradient vector of the image is $[\partial g(x, y)/\partial x, \partial g(x, y)/\partial y]^T$ (i.e. $[g_x, g_y]^T$). Then the gradient direction of a pixel at (x, y) is $\tan\theta = g_y/g_x$.

θ_x and θ_y are denoted the rate of change of θ (i.e. curvature) in x and y direction respectively.

$$\begin{aligned} \theta &= \arctan(g_y/g_x) \\ \frac{\partial\theta}{\partial x} &= \frac{\partial(\frac{g_y}{g_x})/\partial x}{1+(\frac{g_y}{g_x})^2} \\ \theta_x &= \frac{g_{xy}g_x - g_{xx}g_y}{g_x^2+g_y^2} \end{aligned} \tag{3.1}$$

If we differentiate θ with respect to y, we get

$$\theta_y = \frac{g_{yy}g_x - g_{xy}g_y}{g_x^2+g_y^2}. \tag{3.2}$$

Since the gradient vector is perpendicular to the edge, the vector along the edge is $[-g_y, g_x]^T$. Projecting the change of gradient direction vector $[\theta_x, \theta_y]^T$ along the edge, and multiplying the result by the local gradient magnitude, gives the result

$$\begin{aligned} k &= \frac{\theta_x\cdot(-g_y)+\theta_y\cdot(g_x)}{\sqrt{g_x^2+g_y^2}}\cdot\sqrt{g_x^2+g_y^2} \\ &= \frac{g_{xx}g_y^2+g_{yy}g_x^2-2g_{xy}g_xg_y}{g_x^2+g_y^2} \end{aligned} \tag{3.3}$$

Equation (3.3) is applied only at edge points where the Laplacian is zero, because it gives accurate edge position where the corner is located. The approximation comes in the evaluation of g_x, g_y, g_{xy}, g_{xx} and g_{yy}. Theoretically, the continuous function $g(x,y)$ should be differentiated accordingly but it does not exist as a closed form function. As a result, a discrete partial differentiation by Prewitt operators [28] – see Equation (3.4) – is used to obtain the gradient maps of g_x and g_y. By applying Prewitt operators again, $\bar{\Delta}_{3x}$

to g_x, $\bar{\Delta}_{3y}$ to g_y and $\bar{\Delta}_{3x}$ to g_y, g_{xx}, g_{yy} and g_{xy} are obtained respectively.

$$\bar{\Delta}_{3x} = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \qquad \bar{\Delta}_{3y} = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \tag{3.4}$$

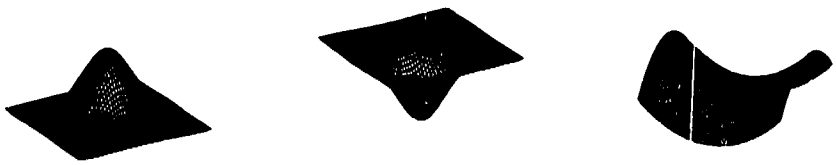
3.1.2 Beaudet's DET

Beaudet [3] applied the Test for Extrema in classical calculus to detect corner points and saddle points. The following is the test definition.

Let $d(x, y)$ be a surface of two variables which has continuous second partial derivatives on a rectangular region Q and let

$$d(x, y) = g_{xx}(x, y)g_{yy}(x, y) - [g_{xy}(x, y)]^2 \tag{3.5}$$

for all (x, y) in Q. The maximum of the absolute value of $d(x, y)$ gives the global maximum, minimum or saddle point in Q – see Figure 3.14.



(a) *(b)* *(c)*

Figure 3.14: *Surface (a) maximum (b) minimum and (c) saddle point*

Wang and Pavlidis [34] categorizes a surface into 11 different topographic features where each type represents an unique appearance in the image. They use these features to recognize and separate written characters. For example,

saddle points occur in places where characters are nearly touching. In our case, corners are found in the global extrema detected by DET.

3.1.3 Facet Model

Zuniga and Haralick [38] fitted a cubic polynomial onto a 5×5 neighborhood where the central pixel must be a Laplacian zero-crossing point (i.e. edge point). Equation (3.6) is the cubic polynomial model,

$$p(x,y) = k_1+k_2x+k_3y+k_4x^2+k_5xy+k_6y^2+k_7x^3+k_8x^2y+k_9xy^2+k_{10}y^3 \quad (3.6)$$

By using the well-known least squares method, we can obtain the coefficients. Since $p(x,y)$ has continuous second partial derivatives, the formula from Kitchen & Rosenfeld and DET can be used to find corners.

$$\begin{aligned}
p_x(x,y) &= k_2 + 2k_4x + k_5y + 3k_7x^2 + 2k_8xy + k_9y^2 \\
p_y(x,y) &= k_3 + k_5x + 2k_6y + k_8x^2 + 2k_9xy + 3k_{10}y^2 \\
p_{xx}(x,y) &= 2k_4 + 6k_7x + 2k_8y \\
p_{xy}(x,y) &= k_5 + 2k_8x + 2k_9y \\
p_{yy}(x,y) &= 2k_6 + 2k_9x + 6k_{10}y
\end{aligned}$$

Furthermore, Zuniga and Haralick derived an equation of the instantaneous rate of change at the central pixel $(0,0)$ of the 5×5 mask. It is exactly the same as Kitchen and Rosenfeld's formula, Equation (3.3), but without the gradient magnitude factor, $\sqrt{g_x^2 + g_y^2}$. That is,

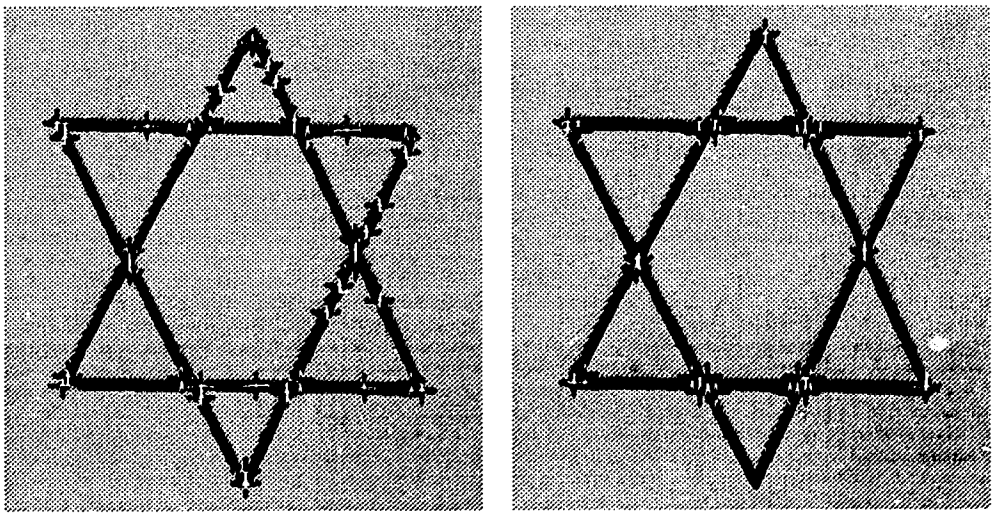
$$k = \frac{-2(k_2^2k_6 - k_2k_3k_5 + k_3^2k_4)}{(k_2^2 + k_3^2)^{3/2}}. \quad (3.7)$$

The -2 coefficient of $k_2^2 k_6$ and $k_3^2 k_4$ are from the differentiation of the cubic polynomial model.

3.1.4 Discussion

All detectors assume the existence of continuous image surface so that the gradient and second derivative can be evaluated. They give accurate and true corner points if the continuous surface assumption is true. However, we know that a real scene image contains many discontinuities such as edges and depth difference, which corrupts the continuity property. Moreover, Deriche and Giraudon [11] stated that Kitchen and Rosenfeld's detector used an unreliable source of information as the base of calculation, namely edge direction. As mentioned in Section 3.1.1, the directional derivatives in x and y direction are approximated by the Prewitt operator, and they are not reliable especially near the points we want to detect: the corners. Moreover, the position of the gradient magnitude maximum is also not reliable, because the maximum point tends to shift away from the true position with different size of angle of the corner. As a result, multiplying the gradient magnitude to the curvature gives a false maximum point — see Figure 3.15(a).

Although Beaudet's operator detects the global extrema (i.e. extrema in all directions), which is easier to find than maxima along a direction, DET suffers a displacement of detected point from the true corner point. The displacement varies with the corner angle and scale in the filtering process but, the locus of the displacement is always along the bisector line of the corner independent of the image contrast. The result is shown in Figure 3.15(b).



(a) *(b)*

Figure 3.15: *Surface fitting corner detection by (a) Kitchen and Rosenfeld (b) Beaudet*

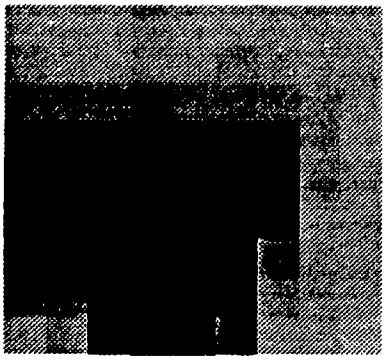
Zuniga and Haralick's facet model provide better continuity environment but the approximate surface is far from representing the actual image. For example, in Figure 3.16(a) shows the original corner image, (b) is the fitted facet with the image on top of it and (c) is the plot of Equation (3.5). The corner point is at the origin and the value of $d(0,0)$ of Equation (3.5) is 3.2566, which is relatively small to its neighbor pixels. This small value is not classified as the extrema of this surface by definition. This totally contradicts our expectation because we thought that DET should give maximum at the corner point. This example shows that a polynomial surface is not adequate in representing the image for DET usage. This is why DET performs badly in Figure 3.17(b). Kitchen and Rosenfeld's formula, Equation (3.3), gives many false corners (see Figure 3.17(a)) due to the wrong gradient magnitude maximum. Once we remove the gradient magnitude factor, then we have the formula of Zuniga and Haralick and the result improves greatly (see Figure 3.17(c)).

3.2 Template matching

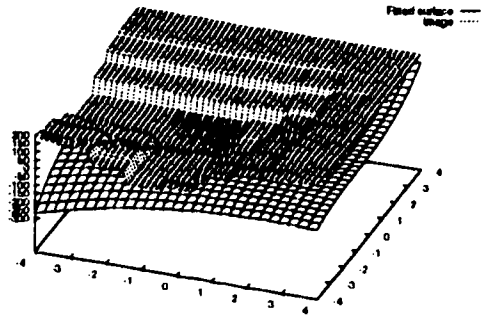
3.2.1 Optimal corner detector

Rangarajan et al. [27] attempted to find the optimal corner operator that should

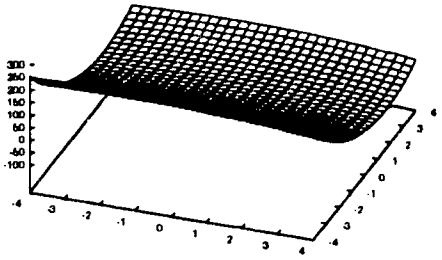
- not be sensitive to noise
- not delocalize the corner points



(a)

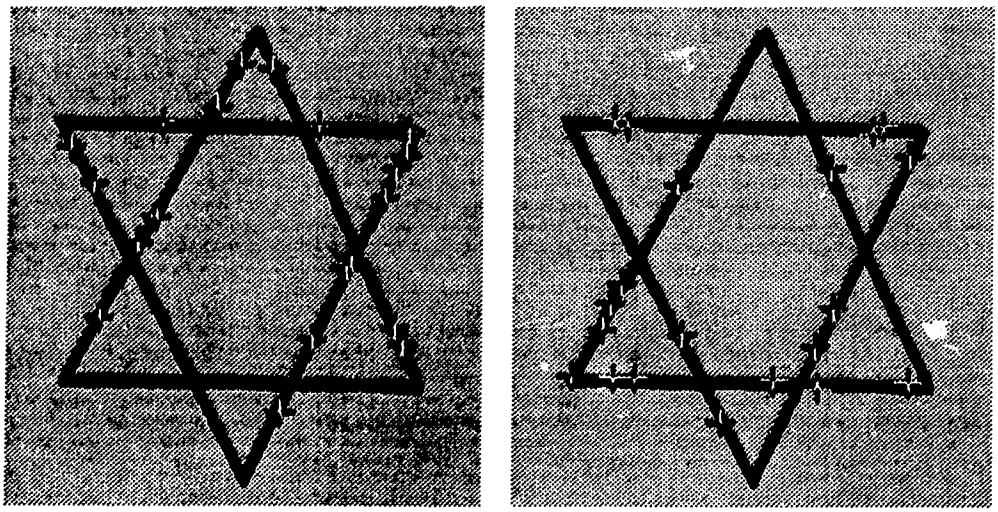


(b)

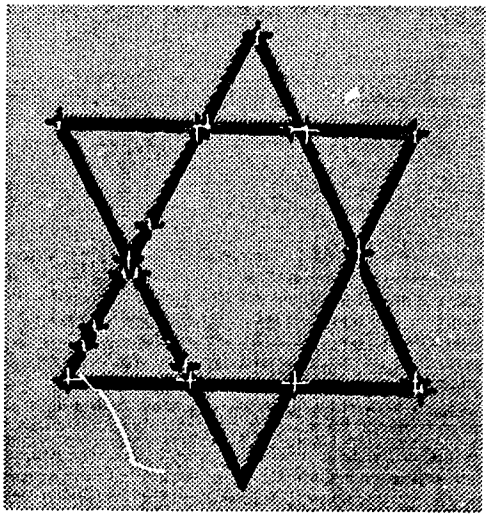


(c)

Figure 3.16: *(a) original corner image (b) fitted surface with image and (c) plot of Test of Extrema of the fitted surface*



(a) *(b)*



(c)

Figure 3.17: *(a) Kitchen and Rosenfeld (b) DET and (c) Zuniga and Haralick corner detector using cubic polynomial model*

- identify corners at some edge points
- detect points that have at least two neighbors which have a different gradient direction than the corner point itself

Figure 3.18 is the corner representation model that they proposed. The shaded area is called the cone portion and the unshaded part is called non-cone portion. The following is the mathematical definition of the model.

$$I(x,y) = \begin{cases} B & \text{if } x > 0 \text{ and } -mx < y < mx \\ 0 & \text{otherwise} \end{cases}$$

An corner operator $g(x,y)$ should reach maximum value at the origin point where the corner is located. The first two criteria can be formalized into

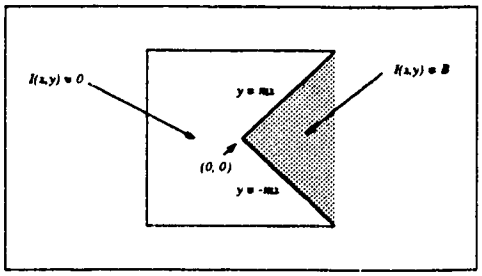


Figure 3.18: *Corner model*

quantitative functions which are Signal to Noise Ratio (SNR) and Delocalization (Λ) of the corner operator $g(x,y)$.

$$\text{SNR} = \frac{B\int_0^{+\infty}\int_{-mx}^{+mx} g(x,y)dydx}{n_0\sqrt{\int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} g^2(x,y)dydx}} \tag{3.8}$$

$$\Lambda = \frac{n_0^2\int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} g_x^2(x,y)dydx}{(B\int_0^{+\infty}\int_{-mx}^{+mx} g_{xx}(x,y)dydx)^2} + \frac{n_0^2\int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} g_y^2(x,y)dydx}{(B\int_0^{+\infty}\int_{-mx}^{+mx} g_{yy}(x,y)dydx)^2} \tag{3.9}$$

where n_0^2 is the variance of the Gaussian noise $n(x,y)$. By maximizing the ratio of SNR to Λ, we can simultaneously satisfy both criteria.

$$\Sigma = \frac{\text{SNR}}{\Lambda}$$

To maximize Σ, SNR has to be maximized and Λ has to be minimized. Rangarajan et al. chose to minimize the $\int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} g^2(x,y)dydx$ term in the denominator of Equation (3.8). By using Lagrange multipliers and Euler equation, they solved the differential equations and obtained two equations.

$$g(x,y) = c_1 \sin\frac{k\pi x}{W}[-(e^{zW} + e^{-zW}) + e^{zy} + e^{-zy}] \tag{3.10}$$

$$g(x,y) = c_2 \sin\frac{n_1\pi x}{W}\sin\frac{n_2\pi y}{W} \tag{3.11}$$

where W is half of the mask side (e.g. 3×3, $W = 1$; 9×9, $W = 4$) and c_1, c_2, k, n_1, n_2, and z are constants. The cone portion of the mask is formed by Equation (3.10) and the noncone portion is formed by Equation (3.11). According to Rangarajan et al. [27], to form a mask for detecting corners in quadrant 1, put the solution of Equation (3.10) (cone portion) in the quadrant 1 region of the mask and put solution of the noncone equation (3.11) in the remaining quadrants. Twelve different detector masks are derived to detect various appearance of corners. Figure 3.19(a) shows that four types of corner may occur in an image in which the cone portion occupies only one quadrant of the mask. Figure 3.19(b) and (c) are corners with cone portion in two quadrants and (d)-(g) are corners that occupy three quadrants. In Rangarajan's et al. first paper [27], they let $k = 1$, making the cone portion negative in quadrant 1. But the solution of Equation (3.10) is inconsistent with the mask they gave, see Figure 3.20(a). None of the people who cited

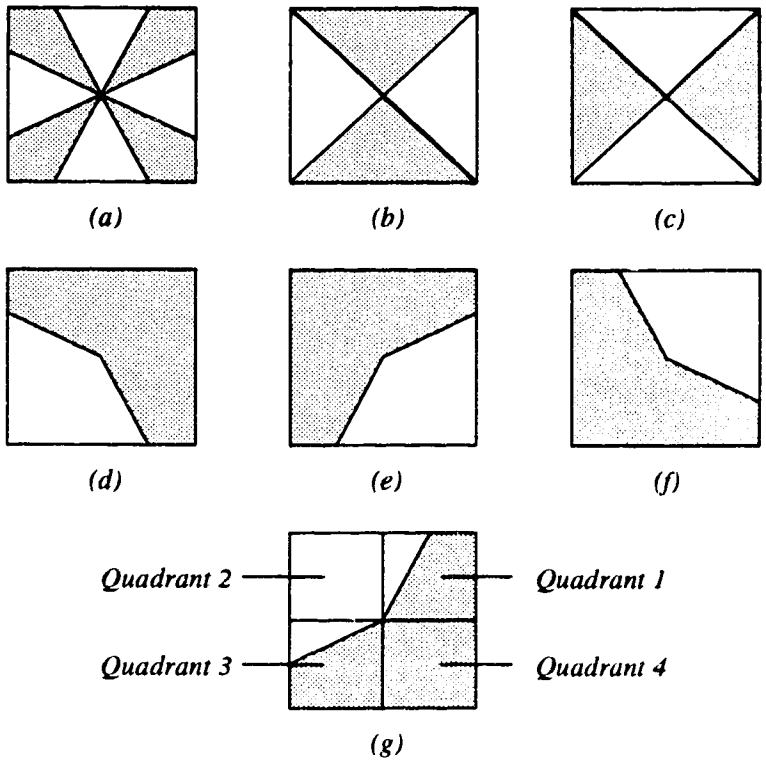


Figure 3.19: *Twelve types of corners*

their work noticed this error. Although later in their journal version, they corrected k to be equal to -1. Even so, it is still impossible to reproduce their mask with their coefficients of $c_1 = 20, c_2 = 20, k = -1, w = 4, z = 0.2, n_1 = 1, n_2 = -1$, see Figure 3.20(b). When Equation (3.11) is plotted

$$\begin{bmatrix} -6 & -11 & -11 & -6 & 0 & 4 & 7 & 7 & 4 \\ -11 & -18 & -18 & -11 & 0 & 8 & 13 & 13 & 8 \\ -11 & -18 & -18 & -11 & 0 & 10 & 17 & 17 & 10 \\ -6 & -11 & -11 & -6 & 0 & 12 & 19 & 19 & 12 \\ 0 & 0 & 0 & 0 & 0 & 12 & 20 & 20 & 12 \\ -6 & -11 & -11 & -6 & 0 & -6 & -11 & -11 & -6 \\ -11 & -18 & -18 & -11 & 0 & -11 & -18 & -18 & -11 \\ -11 & -18 & -18 & -11 & 0 & -11 & -18 & -18 & -11 \\ -6 & -11 & -11 & -6 & 0 & -6 & -11 & -11 & -6 \end{bmatrix}$$

(a)

$$\begin{bmatrix} 2 & 6 & 7 & 4 & 0 & 2 & 4 & 3 & 1 \\ 6 & 15 & 17 & 11 & 0 & 6 & 10 & 9 & 3 \\ 7 & 17 & 19 & 13 & 0 & 9 & 14 & 12 & 4 \\ 4 & 11 & 13 & 8 & 0 & 11 & 16 & 14 & 6 \\ 0 & 0 & 0 & 0 & 0 & 11 & 17 & 15 & 6 \\ -4 & -11 & -13 & -8 & 0 & 8 & 13 & 11 & 4 \\ -7 & -17 & -19 & -13 & 0 & 13 & 19 & 17 & 7 \\ -6 & -15 & -17 & -11 & 0 & 11 & 17 & 15 & 6 \\ -2 & -6 & -7 & -4 & 0 & 4 & 7 & 6 & 2 \end{bmatrix}$$

(b)

Figure 3.20: *Rangarajan's mask (a) and my mask (b)*

(see Figure 3.21), I found that it is not possible to obtain their result because both quadrant 2 and 4 are positive which is opposite to Rangarajan et al's. mask. I noticed that they did not follow their described method [27] but they

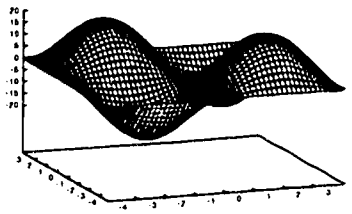


Figure 3.21: *Plots of Noncone equation*

replicated the negative part of noncone portion in the quadrants that corner will not appear. By using this method, I generate a mask for quadrant 1, Figure 3.22. In my implementation, masks are convolved with edge points

$$\begin{bmatrix} -2 & -6 & -7 & -4 & 0 & 2 & 4 & 3 & 1 \\ -6 & -15 & -17 & -11 & 0 & 6 & 10 & 9 & 3 \\ -7 & -17 & -19 & -13 & 0 & 9 & 14 & 12 & 4 \\ -4 & -11 & -13 & -8 & 0 & 11 & 16 & 14 & 6 \\ 0 & 0 & 0 & 0 & 0 & 11 & 17 & 15 & 6 \\ -4 & -11 & -13 & -8 & 0 & -8 & -13 & -11 & -4 \\ -7 & -17 & -19 & -13 & 0 & -13 & -19 & -17 & -7 \\ -6 & -15 & -17 & -11 & 0 & -11 & -17 & -15 & -6 \\ -2 & -6 & -7 & -4 & 0 & -4 & -7 & -6 & -2 \end{bmatrix}$$

Figure 3.22: *Revised mask*

only in order to satisfy the third criterion stated at the beginning. The fourth condition has not been met explicitly because of the unreliability of the gradient direction.

3.2.2 Förstner's interest operator

Förstner [14, 15] took another approach to find feature points in an image. He called this detector an interest operator, because not only can it detect corners but it also finds distinct points. He looked into three different tasks which can be written as a least squares problem in the form of Gauss-Markoff model.

$$\vec{x} + \vec{e} = \boldsymbol{A}\vec{y} \tag{3.12}$$

$$\boldsymbol{\Sigma} = \sigma_0^2 \boldsymbol{W}^{-1} \tag{3.13}$$

where $\vec{x}$ contains m observed values, $\vec{e}$ is the errors, $\vec{y}$ contains the u unknowns, $\boldsymbol{A}$ is the full rank $m \times u$ design matrix, $\boldsymbol{\Sigma}$ is the covariance matrix, $\boldsymbol{W}$ is called the weight matrix and σ_0, by definition, is an arbitrary constant, called the variance factor. The term "weight" is used to express precision by way of inverse relationship. Thus high weight means high precision, which

in turn means small standard deviation.

$$\boldsymbol{\Sigma} = \begin{bmatrix} \sigma_1^2 & \sigma_{12} & \cdots & \sigma_{1m} \\ \sigma_{21} & \sigma_2^2 & \cdots & \sigma_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{m1} & \sigma_{m2} & \cdots & \sigma_m^2 \end{bmatrix}$$

The normal equation method is used to solve this least square problem.

$$\boldsymbol{N}\vec{y} = \vec{h}$$

where $\boldsymbol{N}$ is the normal matrix with

$$\boldsymbol{N} = \boldsymbol{A}^T\boldsymbol{W}\boldsymbol{A}, \quad \vec{h} = \boldsymbol{A}^T\boldsymbol{W}\vec{x} \tag{3.14}$$

With this model, three tasks are described below. All they have in common is that the only unknowns are the coordinates of the interest points (x_0, y_0).

Task 1: Corner detection - intersection of edge elements

Each pixel is regarded as an edge element with an orientation derived from the gradient of gray levels $\nabla^T g(x,y) = [g_x(x,y), g_y(x,y)]$. A corner (x_0, y_0) can be estimated from the intersection of all the edge elements inside the window. A straight line through (x,y) can be represented by $x\cos\theta + y\sin\theta - l = 0$, (see Figure 3.23) where l is the distance of the origin from the line and θ is the gradient direction (normal direction) of the line. e is the error distance of intersection between the line and point (x_0, y_0). The general linear model for the intersection point is

$$l_i + e_i = \cos\theta_i x_0 + \sin\theta_i y_0 \quad \text{for } i = 1 \ldots m \tag{3.15}$$

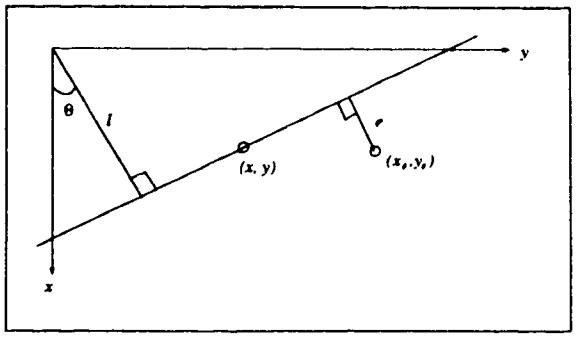


Figure 3.23: *Line model*

This model can be formulated as a least square problem like Equation (3.12) where

$$\vec{x} = \begin{bmatrix} l_1 \\ l_2 \\ \vdots \\ l_m \end{bmatrix} \qquad \vec{e} = \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_m \end{bmatrix} \tag{3.16}$$

$$\boldsymbol{A} = \begin{bmatrix} \cos\theta_1 & \sin\theta_1 \\ \cos\theta_2 & \sin\theta_2 \\ \vdots & \vdots \\ \cos\theta_m & \sin\theta_m \end{bmatrix} \qquad \vec{y} = \begin{bmatrix} x_0 \\ y_0 \end{bmatrix}. \tag{3.17}$$

We assume the variance of greylevel noise to be σ_n^2 and constant. Moreover, the standard deviation of l is inversely proportional to the strength of the edge (i.e. $\sigma_l = \sigma_n/|\nabla g|$), in other words, the stronger the edge is, the more precise of the approximation will be. Since the noise is constant and noncorrelated, $\boldsymbol{\Sigma}$ is a diagonal matrix with $\sigma_{ii} = \sigma_l$. In this case, it is reasonable to assign the arbitrary constant σ_0 to σ_n in order to cancel the factor σ_n on the left

hand side of Equation (3.13). And so from Equation (3.13), we get

$$\boldsymbol{W} = |\nabla g|^2 \boldsymbol{I}. \tag{3.18}$$

Substitute Equation (3.16), (3.17) and (3.18) into (3.14) we obtain

$$\boldsymbol{N} = \begin{bmatrix} \cos\theta_1 & \cos\theta_2 & \cdots & \cos\theta_m \\ \sin\theta_1 & \sin\theta_2 & \cdots & \sin\theta_m \end{bmatrix} |\nabla g|^2 \boldsymbol{I} \begin{bmatrix} \cos\theta_1 & \sin\theta_1 \\ \cos\theta_2 & \sin\theta_2 \\ \vdots & \vdots \\ \cos\theta_m & \sin\theta_m \end{bmatrix}$$

and since

$$\cos\theta = \frac{g_x}{|\nabla g|}, \qquad \sin\theta = \frac{g_y}{|\nabla g|}$$

then

$$\boldsymbol{N} = \begin{bmatrix} \sum_1^m g_x^2 & \sum_1^m g_x g_y \\ \sum_1^m g_x g_y & \sum_1^m g_y^2 \end{bmatrix}. \tag{3.19}$$

Task 2: Least square matching

Suppose $g_0(x, y)$ represents an object $g(x, y)$ shifted by (x_0, y_0) and with added noise $n(x, y)$.

$$g_0(x, y) = g(x + x_0, y + y_0) + n(x, y) \tag{3.20}$$

which, by using first-order Taylor series, can be approximated as,

$$g_0(x, y) - g(x, y) - n(x, y) \approx g_x(x, y)x_0 + g_y(x, y)y_0$$

which again can be written in the form like Equation (3.12) where

$$\vec{x} = \begin{bmatrix} dg(x_1, y_1) \\ dg(x_2, y_2) \\ \vdots \\ dg(x_m, y_m) \end{bmatrix} \qquad \boldsymbol{A} = \begin{bmatrix} g_x(x_1, y_1) & g_y(x_1, y_1) \\ g_x(x_2, y_2) & g_y(x_2, y_2) \\ \vdots & \vdots \\ g_x(x_m, y_m) & g_y(x_m, y_m) \end{bmatrix} \tag{3.21}$$

where $dg(x_i, y_i) = g_0(x_i, y_i) - g(x_i, y_i)$. Because the variance of dg, σ_{dg}, is identical to that of the noise σ_n, $\boldsymbol{\Sigma}$ again is a diagonal matrix with constant value of σ_n. As a result, according to Equation (3.13), σ_0 is the standard deviation of the noise σ_n and thus $\boldsymbol{W}$ is the identity matrix. By knowing $\boldsymbol{W}$, we can derive the normal matrix $\boldsymbol{N}$, by substituting Equation (3.21) into (3.14),

$$\boldsymbol{N} = \begin{bmatrix} g_x(x_1, y_1) & g_x(x_2, y_2) & \cdots & g_x(x_m, y_m) \\ g_y(x_1, y_1) & g_y(x_2, y_2) & \cdots & g_y(x_m, y_m) \end{bmatrix} \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} g_x(x_1, y_1) & g_y(x_1, y_1) \\ g_x(x_2, y_2) & g_y(x_2, y_2) \\ \vdots & \vdots \\ g_x(x_m, y_m) & g_y(x_m, y_m) \end{bmatrix} \tag{3.22}$$

$$\boldsymbol{N} = \begin{bmatrix} \sum_1^m g_x^2(x, y) & \sum_1^m g_x(x, y) g_y(x, y) \\ \sum_1^m g_x(x, y) g_y(x, y) & \sum_1^m g_y^2(x, y) \end{bmatrix}. \tag{3.23}$$

Task 3: Weighted centre of gravity

Each pixel in a window contributes to the centre of gravity of that window by using the gradient as weight. The linear model is

$$\begin{aligned} x + e_x &= x_0 \\ y + e_y &= y_0 \end{aligned}$$

where e_x and e_y are the errors in the x and y direction. From this model, the design matrix, $\boldsymbol{A}$, is an identity matrix. The weight of each pixel depends on the direction of its local gradient. In order to rotate the vector $[|\nabla g|, 0]$ into $\nabla g(x, y)$'s direction, the rotation matrix is used

$$\boldsymbol{R}_\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tag{3.24}$$

where θ is the direction of gradient ∇g. Applying the propagation of weight matrices [25], we obtain

$$\boldsymbol{W} = |\nabla g|^2 \begin{bmatrix} \cos^2\theta & \cos\theta\sin\theta \\ \cos\theta\sin\theta & \sin^2\theta \end{bmatrix}$$

which gives the same normal equation matrix as Equation (3.19).

Equation (3.25) is the normal equation of least square and Equation (3.26) is for corner detection and weighted gravity. Note that the normal equation of corner and weighted gravity are identical to each other which means they are the same operation but under different interpretation.

$$\begin{bmatrix} \sum_1^m g_x^2 & \sum_1^m g_x g_y \\ \sum_1^m g_x g_y & \sum_1^m g_y^2 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} = \begin{bmatrix} \sum_1^m g_x dg \\ \sum_1^m g_y dg \end{bmatrix} \tag{3.25}$$

$$\begin{bmatrix} \sum_1^m g_x^2 & \sum_1^m g_x g_y \\ \sum_1^m g_x g_y & \sum_1^m g_y^2 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} = \begin{bmatrix} \sum_1^m g_x^2 x + \sum_1^m g_x g_y y \\ \sum_1^m g_x g_y x + \sum_1^m g_y^2 y \end{bmatrix} \tag{3.26}$$

where the sum includes all pixels in the window. After forming the normal equation matrix $\boldsymbol{N}$, the unknown parameters x_0 and y_0 that are the coordinate of the interest point can be estimated. Whether a point is distinct is

determined by the size w and the roundness q of the confidence ellipse of the estimation. According to Förstner [15], they are defined as:

$$w = \frac{1}{trace(\boldsymbol{N}^{-1})} = \frac{det\boldsymbol{N}}{trace(\boldsymbol{N})} \tag{3.27}$$

$$q = \frac{4det(\boldsymbol{N})}{(trace(\boldsymbol{N}))^2} = \frac{4w}{trace(\boldsymbol{N})} \tag{3.28}$$

The parameters w and q are called interest values. Points that have greater values than w_{lim} and q_{lim} are selected as interest points. The roundness measure q lies in the range between 0 and 1. If $q = 1$, the error ellipse is a circle. If $q = 0$, the window is lying on an ideal edge. As a result, in order to detect distinct points, q has to be greater than 0.5. Experiments by Förstner suggested that values q_{lim} between 0.5 and 0.75 work well. These values correspond to ratio 2 and $\sqrt{3}$ of the semiaxes of the error ellipse.

The threshold w_{lim} is to suppress windows containing only flat areas. This threshold should be made dependent on the global image content. For example, it could be the average w_{mean} of all the weights, or the medium w_{med} of the weights taken over the whole image.

$$w_{lim} = \begin{cases} f \cdot w_{mean} \\ c \cdot w_{med} \end{cases}$$

where f is between 0.5 and 1.5 or $c = 5$ for best results according to Förstner [15]. Finally, all local non-maxima must be suppressed. The suppression window has a size of 7×7. Only those interest points with largest weight size are retained.

In my implementation, the Robert gradient operator [23] is used to obtain the gradient value of the 5×5 window – see Figure 3.24.

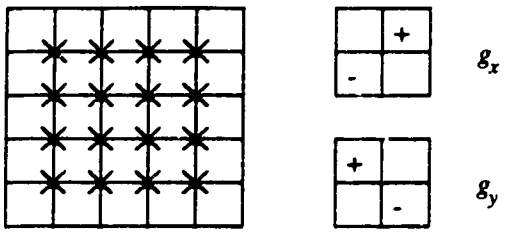


Figure 3.24: *Robert gradient operator in a 5 × 5 window*

3.2.3 Lü's interest operator

Lü [23] developed a much simpler operator to detect interest points in three simple steps:

1. Ground operator: A point is selected if at least two of the absolutes of the gradients are larger than a threshold d. Lü obtained the best d value experimentally with value between 5 and 10 with 3×3 local average filtering and between 8 to 15 without any filtering.

2. Points selected from first step are used to calculate their interest value by summing the absolute gradient values of the eight neighbor pixels.

3. Suppress all local non-maxima with a 7×7 window.

3.2.4 Förstner and Lü combined interest operator

Lü [23] combined his ground operator with the q evaluation by Förstner to create a fast and accurate interest operator. First of all, points are selected by the ground operator and then the q value of the Förstner operator is calculated as an interest value. Points that have q greater than 0.5 are

selected as interest points. Finally, all interest points are filtered by a local non-maxima suppression process to obtain the final feature points.

3.2.5 Discussion

Although I cannot obtain the same mask as that given in Rangarajan's paper, I discovered that placing the positive cone portion in different quadrants can achieve satisfying results. I notice that in order to respond to corners, positive weight must be present in at least one quadrant. I have tried my masks and deliberately placed the positive weight in the cone quadrant. And the result is shown in Figure 3.25. Rangarajan stated that single cone portion masks

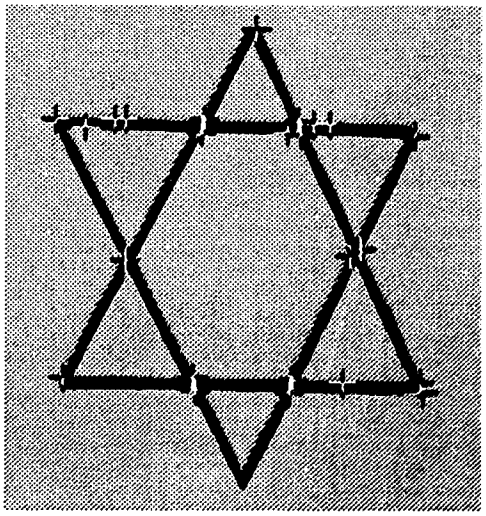


Figure 3.25: *Corner detection with* 9×9 *optimal detector*

have maximum response at right angle (i.e. 90°). In the same way, double cone portion masks have maximum response at 180° angle — a straight line.

Figure 3.25 shows that some false corners on the straight lines are detected. These masks are less sensitive to small angle than angles close to 180°. This lack of sensitivity makes it miss many corners.

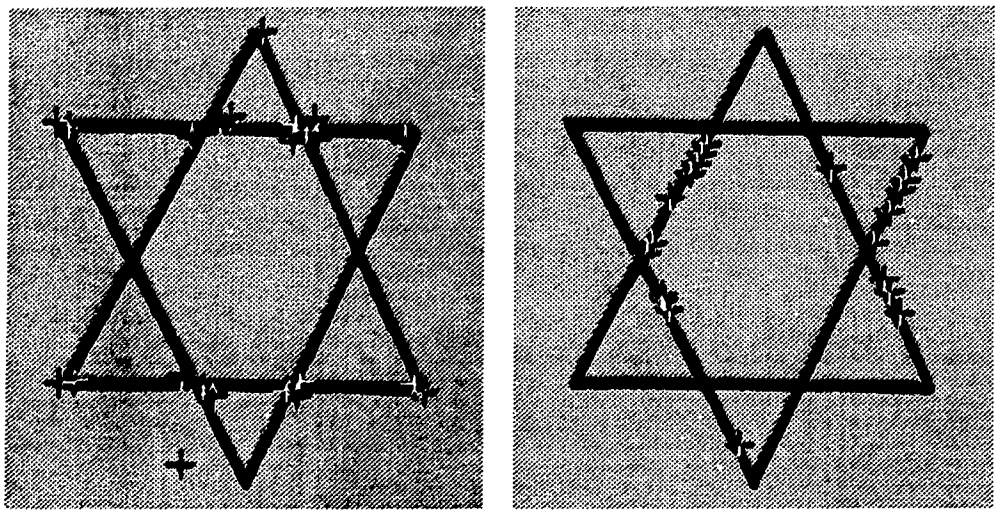
Förstner's interest operator focuses on corners and least square matching. Although Förstner used the unreliable gradient direction as the base of calculation, he used the gradient direction from all pixels in the window and approximated their intersection by the least square method. It is different from Kitchen and Rosenfeld's method because balancing the error over the whole window can minimize the error caused by a single pixel. Moreover, Förstner's interest operator is more sensitive to image texture than Rangarajan's masks because Förstner applied formulas to the window pixels at each new position instead of using the same masks all the time. As a result, the interest value is always directly derived from the image information. Furthermore, since the determination of interest points are by evaluating the size and roundness of the error ellipse, it is more adaptive to the image than Rangarajan's fixed masks. However, problems may arise in determining the threshold w_{lim} because Förstner found that w_{lim} depends in an unpredictable way on the edge content, the sharpness of the edges and the noise level in the image. Although a better solution was suggested which is to take the median of weights over the whole image, the computation size is two times the number of pixels in the image. The result is shown in Figure 3.26(a). Due to this major drawback, Lü developed a fast interest operator which involves only a few simple gradient calculations. However, because of its excessive simplicity, the quality of feature detection is low. The result is shown in Figure 3.26(b). However, combining Lü's ground operator with Förstner's

error ellipse estimation gives a better result. Lü's ground operator eliminates the trouble of evaluating w_{lim} and serves the purpose of w_{lim} because points that are selected by the ground operator are not in flat areas. The combined version still preserves all the advantages of Förstner's operator and enhances computational efficiency. The result is shown in Figure 3.26(c).

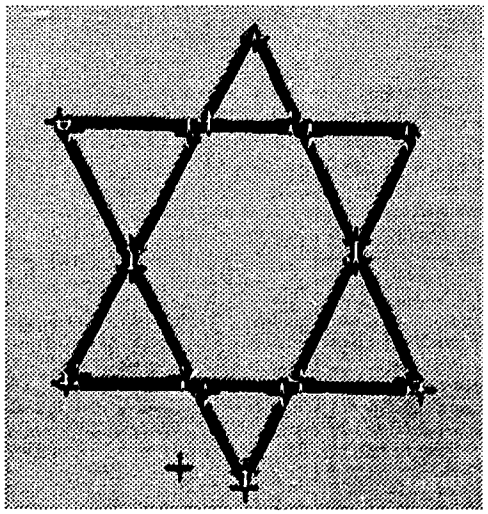
3.3 Experimental results

To determine which method is the best, a series of experiments were performed on two image pairs (I_1, I_2) and (I_3, I_4). All images are captured by the COHU camera (i.e. noise is present) and paired images are viewing the same object but at slightly different angle (50 points of panning). The sizes of Figure 3.27(a), (b), (c) and (d) are 176×225, 175×224, 183×207 and 181×208 respectively. The solid objects in (I_1, I_2) has twelve well-defined corners and the line objects (I_3, I_4) have thirty-six well-defined corners in them. Two types of object are used because we want to see the robustness of the corner detectors with different type of corners.

The criteria of evaluation are stability, speed and interpretability (see definition in page 37). In feature based matching, stability is the most important condition because the point correspondence algorithm can perform at best when a large number of the same points appears in both images. Therefore, the merit of a corner detection algorithm is determined in the following priority: stability, speed and interpretability. In Table 3.2 and 3.3, the first column contains names of each method, second and third columns show the number of points extracted. One should note that not all extracted



(a) *(b)*



(c)

Figure 3.26: *Feature extraction by (a)Förstner (b)Lü interest operator and (c)Lü & Förstner combined version*

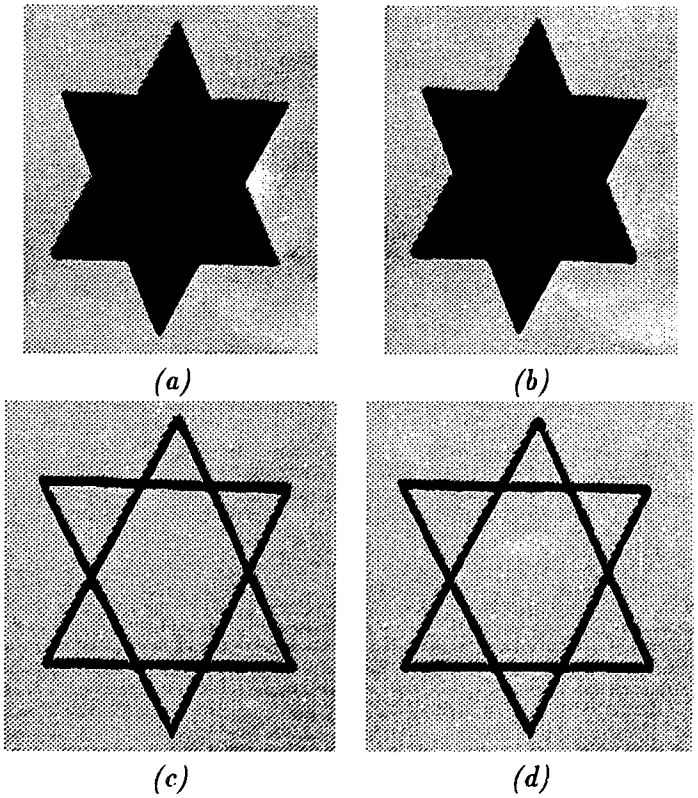


Figure 3.27: *Test images (a) I_1, (b) I_2 (c) I_3 and (d) I_4*

Method	Corners in I_1	Corners in I_2	Matched points	Run time I_1 (sec.)	Run time I_2 (sec.)	Detected corners
K & R	12	13	12/13	3.0	3.0	12/12
DET	11	12	11/12	2.6	2.6	11/12
Facet	14	11	11/14	12.4	12.4	11/12
Rangarajan	8	11	3/11	8.5	8.5	4/12
Förstner	14	15	9/15	7.0	7.0	7/12
Lü	15	15	3/15	3.7	3.7	2/12
F & L combo	15	15	10/15	2.3	2.3	10/12

Table 3.2: *Experimental results with solid object*

Method	Corners in I_1	Corners in I_2	Matched points	Run time I_1 (sec.)	Run time I_2 (sec.)	Detected corners
K & R	37	30	20/37	2.8	2.8	21/36
DET	24	20	18/24	2.4	2.4	24/36
Facet	27	28	14/28	12.3	12.3	15/36
Rangarajan	24	19	12/24	8.3	8.3	17/36
Förstner	29	26	18/29	6.4	6.4	24/36
Lü	40	40	12/40	3.5	3.5	6/36
F & L combo	23	21	20/23	1.27	1.27	21/36

Table 3.3: *Experimental results with line object*

points are well-defined corners — some of them are falsely detected. Each corner detector limits its output to only a maximum of fifteen points for a solid object and forty points for a line object. The fourth column contains the stability ratio where the numerator is the number of same points appear in both images and the denominator is the maximum number of feature points extracted from either image. The next two columns shows the time taken to do the detection. All processes were run on a SunSparc 1 and each timing is the average of five runs, to average out system load fluctuations. The last column is the interpretability ratio which is the number of detected well-defined corners over the number of actual well-defined corners. The best method should have high stability, high interpretability ratio and short run time.

In the solid object experiment (Table 3.2), all methods perform well except Rangarajan's, Förstner's and Lü's detectors. The problem of Rangarajan's detector is that their corner model assumes a positive corner (i.e. intensity of cone portion is higher than noncone portion), but all the corners are negative in this experiment. This is why it cannot detect corners well. Lü's detector is expected to do badly on corner detection because it basically is a distinct point finder. A distinct point, according to Lü, means a pixel that has certain higher intensity than its neighbor pixels. This surely cannot detect a corner. On the other hand, Förstner's detector is supposed to be good at corner detection, because the threshold w_{lim} varies depending on the edge content. A solid object, which has few edges, makes the determination of w_{lim} very hard. We use $0.5w_{mean}$ as the threshold but it still gives unsatisfactory results. This shows that the Förstner interest operator is not suitable

for fully automatic feature detection.

In the line object experiment (Table 3.3), the results deteriorate because a line object is harder to handle than solid object with homogeneous intensity. Methods that do well in the first experiment give poorer results, while the poor ones give better results this time. For example, the Facet model and Kitchen & Rosenfeld's detector have less stability because line objects deviate greatly from their assumed smooth surface. DET keeps its high performance because it is measuring the global extrema which is not seriously affected by the line object. Rangarajan's detector gives good results because a line object has both positive and negative corners. As a result, Rangarajan's detector finds all the positive corners. Förstner's interest operator does well because there are plenty of edges in the image that makes the thresholding valid.

From the experiments shown above, DET and the Förstner & Lü combo have the highest stability, shortest run time and highest interpretability among two types of object. However, the Förstner & Lü combo are susceptible to noise because of the primitive ground operator. As a result, I deduce that DET is preferable for use of the next phase — the point matching algorithm.

Chapter 4

Point Matching Algorithms

After a set of feature points has been extracted from each image, the next task is to find the point correspondence between the two images. Many techniques have been developed to handle this problem. The simplest and most intuitive method is the brute-force point-by-point matching using gray-level correlation. But the computation of correlation is very expensive. Another approach of point matching is to use the similarity of the relative distance among the control points in an image. Instead of comparing the image intensity of each feature point, the distances between feature points within an image become the key of matching. It has an advantage of using more global information. Global information is a natural need for good matching because when we try to compare two images, we look for a large object or pattern in the images as the key for matching. That means if an algorithm can make use of global information, better matching is expected. Multiresolution matching is aiming at this specific goal. By reducing image size, one can analyze the

image faster yet cover a large portion of the image. As it iterates upward in the resolution hierarchy, the accuracy of matching improves. Finally, if feature extraction is not reliable, in other words, the feature points between images are not stable, we have to force the point correspondence in order to match the images. This can be done by defining a grid in each image and find out the movement of grid points in one image with respect to the other. Since the point correspondence is inherent from the grid formation, the point matching task is automatically done. As a result, the main task of this approach is to find the correct grid points movement.

In this chapter, three approaches described above are explored: grid point matching, relative distance matching using relaxation and multiresolution matching. Their advantages and disadvantages are discussed to determine which one to incorporate into the registration system. But before we explore these algorithms, a fundamental component of them must be studied: The similarity measure. Similarity measure is fundamental because it measures the goodness of match. The merit of the point matching algorithm depends greatly on the reliability of the similarity measure. Therefore, the first section of this chapter is the study of similarity measure and then we will carry the knowledge of this study to the following point mapping algorithms.

4.1 Similarity Measure

Suppose $S(x, y)$ is a $M \times N$ image and $T(x, y)$ is a $K \times L$ template to be matched against $S(x, y)$. Four similarity measure methods are presented: Normalized Correlation (NC), Sum of Absolute Valued Differences (SAVD),

Stochastic Sign Change (SSC) and Intensity Combinatorial Minimization (ICM).

4.1.1 Normalization Correlation

The normalized correlation coefficient function [6], henceforth referred to as the correlation coefficient, has traditionally been a popular choice. The goodness of fit between pixels $S(x,y)$ and $T(u,v)$ is defined as

$$\psi_{ST}(x,y,u,v) = \frac{1}{\sigma_S \sigma_T KL} \sum_{i=-(L-1)/2}^{(L-1)/2} \sum_{j=-(K-1)/2}^{(K-1)/2} (S(x+i,y+j)-\mu_S)(T(u+i,v+j)-\mu_T) \tag{4.1}$$

where

$$\begin{aligned}
\mu_S &= \frac{1}{KL} \sum_{i=-(L-1)/2}^{(L-1)/2} \sum_{j=-(K-1)/2}^{(K-1)/2} S(x+i,y+j) \\
\mu_T &= \frac{1}{KL} \sum_{i=-(L-1)/2}^{(L-1)/2} \sum_{j=-(K-1)/2}^{(K-1)/2} T(u+i,v+j) \\
\sigma_S &= \sqrt{\frac{1}{KL-1} \sum_{i=-(L-1)/2}^{(L-1)/2} \sum_{j=-(K-1)/2}^{(K-1)/2} S(x+i,y+j)^2 - \mu_S^2} \\
\sigma_T &= \sqrt{\frac{1}{KL-1} \sum_{i=-(L-1)/2}^{(L-1)/2} \sum_{j=-(K-1)/2}^{(K-1)/2} T(u+i,v+j)^2 - \mu_T^2}
\end{aligned}$$

The maximum point indicates the best match.

4.1.2 Sum of Absolute Valued Differences

The sum of absolute valued differences (SAVD) [2] is computationally more economical than the correlation coefficient. It is formulated as follows:

$$\xi_{ST}(x,y,u,v) = \sum_{i=-(L-1)/2}^{(L-1)/2} \sum_{j=-(K-1)/2}^{(K-1)/2} |T(u+i,v+j) - S(x+i,y+j)| \quad (4.2)$$

The best fitted point should be the minimum point.

4.1.3 Stochastic Sign Change

Suppose that two images $I_1(x,y)$ and $I_2(x,y)$ differ only because of noise measurement which is assumed to be additive, zero mean, with a symmetric density function. Let $D(x,y) = I_1(x,y) - I_2(x,y)$ be the difference image. The stochastic sign change (SSC) criterion [33] is defined as the number of sign changes in the sequence of values of $D(x,y)$ scanned line by line or column by column (e.g. in the sequence $--+-+-$ there are 4 sign changes). Maximum sign changes indicates best fitted position.

4.1.4 Intensity Combinatorial Minimization

Mustafa and Ganter [26] introduced a new metric for similarity defined as follows:

$$R(x,y,u,v) = \sum_{i=-(L-1)/2}^{(L-1)/2} \sum_{j=-(K-1)/2}^{(K-1)/2} \Gamma(T(u+i,v+j), S(x+i,y+j))$$

W is a set containing all the unique combinations and initially it is empty $W = \emptyset$, then

$$\Gamma(x,y) = \begin{cases} 1 \text{and } (x,y) \in W & \text{if } (x,y) \notin W \\ 0 & \text{if } (x,y) \in W \end{cases}$$

for the best matched point

$$\begin{aligned} (x_0, y_0) &= \{(x,y) | \min\{R(x,y)\}\} \\ &\text{for } 1 \leq x \leq M - K + 1, 1 \leq y \leq N - L + 1 \end{aligned}$$

with $M \times N$ image and $K \times L$ template. This method counts the number of distinct pixel pairs in the images and chooses the smallest. The method is called intensity combinatorial minimization (ICM).

4.1.5 Experimental results

Three sets of experiment are conducted to determine which similarity measure method can find the best match under different circumstances. The experiments are simply sliding a template extracted from the original image and finding the position where it is croped out. Template size, noise level and percentage of flat area are the three variables that reflect the performance and robustness of these similarity measure methods. Firstly, template size affects the amount of computation and accuracy of matching. The time needed to do the matching is directly proportional to the template size. As the size decreases, which means less information is used, computation is shorter but the quality of measure deteriorates. In the experiment, five different sizes: $64 \times 64, 32 \times 32, 16 \times 16, 8 \times 8$ and 4×4 of template are used to see the effect of

sizing. Secondly, noise always corrupts the accuracy of measurement. Gaussian noise with $\sigma = 10, 20, 30, 40$ and 50 is added to the image to observe which measure is the most robust to noise. Lastly, when flat areas dominate the template area, false matching usually results. A flat area is a region that the maximum difference of gray-level intensity among the pixels is less than 10. Selected templates with 20%, 40%, 60%, 80% and 100% of flat areas are used for the test. Table 4.4, 4.5 and 4.6 are the results and Figure 4.28 shows the relationship of template size and time.

Size	NC	SAVD	SSC	ICM
64×64	0	0	0	0
32×32	0	0	0	0
16×16	0	0	17.2	80.2
8×8	0	0	16.5	14.4
4×4	12.0	10.0	47.9	29.4

Table 4.4: *Distance from the correct position with various template sizes,* $\sigma = 30$ *and* $FA\% = 60$

σ	NC	SAVD	SSC	ICM
10	0	0	0	0
20	0	0	0	0
30	0	0	0	12.0
40	0	0	0	14.4
50	0	0	0	8.0

Table 4.5: *Distance from the correct position with different noise level, template size* $= 16 \times 16$ *and* $FA\% = 60$

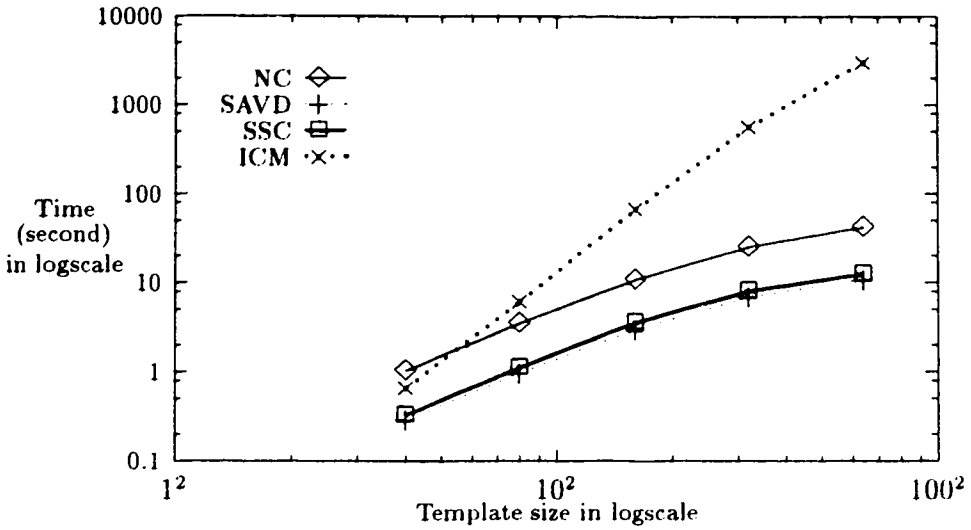


Figure 4.28: *Time performance with various template sizes*

FA %	NC	SAVD	SSC	ICM
20	0	0	0	36.0
40	0	0	0	11.0
60	0	0	14.0	22.0
80	0	0	1.41	31.8
100	53.3	36.2	19.2	27.9

Table 4.6: *Distance from the correct position with different percentage of flat area,* $\sigma = 30$ *and template size* $= 16 \times 16$

4.1.6 Discussion

From Table 4.4, all methods fail when a 4×4 template is used because there is too little information in a template to distinguish correct and false matches. SSC and ICM start failing to match with the size of 16×16 and 8×8 respectively. This indicates that both methods need much more information. Note that once a method fails, the detected match point is random. This phenomenon can be seen in ICM. For ICM, since it is counting the number of greylevel combinations between the image and template, the maximum number of distinct combinations for a 4×4 template is only 16, which means that from an 80×80 image ICM can possibly find $80 \times 80/16$ matched points. If the template size increases, more distinct combination exists which means that matched points are more unique. SSC depends on the count of sign change. It is similar to ICM that the maximum possible sign change is 15 which is even smaller than ICM.

Figure 4.28 shows the relationship between computation time and template size. The y-axis is in logarithmic scale because the range of ICM is too large, from 0.67 to 3060.4 seconds. The reason for x-axis to be in log scale is to show the linear property of ICM in this scale. We can use linear regression method to approximate the complexity of all methods. The linear model is as follows,

$$\begin{aligned} Y &= mX + c \\ \ln y &= m \ln x + c \\ \ln y - m \ln x &= c \end{aligned}$$

$$\begin{aligned} \ln(\frac{y}{x^m}) &= c \\ y &= e^c x^m. \end{aligned}$$

where m and c are the slope and y-axis intersection respectively. For ICM, c and m are -4.54 and 3.08 respectively which means ICM is an $O(n^3)$ algorithm where n is the size of the template. The c and m values of NC, SAVD and SSC are $(-1.63, 1.36), (-2.85, 1.33)$ and $(-2.72, 1.34)$ respectively. They are nearly linear to template size because only one-pass is required every time.

The results of the noise test are shown in Table 4.5. NC, SAVD and SSC are insensitive to noise in the image but ICM cannot handle noise with $\sigma = 30$ or greater. This is because noise increases the number of combinations unpredictably. On the other hand, since the added noise is equally distributed and has zero mean, it does not affect the performance of NC, SAVD and SSC. In the calculation of NC, zero mean noise is mostly eliminated in μ_S, μ_T, σ_S and σ_T evaluations, see Equation (4.1). For SAVD, the formula is

$$\begin{aligned} SAVD &= \sum_{i=-(L-1)/2}^{(L-1)/2} \sum_{j=-(K-1)/2}^{(K-1)/2} |T(u+i, v+j) - (S(x+i, y+j) + n(x+i, y+j))| \\ &= SAVD_{old} + \sum_{i=-(L-1)/2}^{(L-1)/2} \sum_{j=-(K-1)/2}^{(K-1)/2} |n(x+i, y+j)| \end{aligned}$$

where n is the added noise. Because the noise is equally distributed, the sum $\sum\sum |n(x,y)|$ is constant which means the point possessing the minimum SAVD with no noise is still the minimum point even when noise is present. SSC is designed based on the assumptions of zero-mean and equally distributed noise [33] so that it can withstand a high level of noise disturbance.

Lastly, the proportion of flat area in the template also greatly affects the performance especially ICM. As shown in Table 4.6, ICM cannot have a correct registration in all trials. This is because the noise level is too high for it to perform well. Moreover, all methods fail when the template is just a flat plane. Because noise is added to a flat area, the sign-to-noise ratio is low and that deteriorates all the measurements.

In summary, ICM is the weakest method because it depends on the count of intensity combinations which is highly susceptible to noise and size of the template. SSC's performance is not as stable as NC and SAVD. It fails when template is less than 16×16 and flat area percentage is greater than 60%. Both NC and SAVD are equally robust in template size, noise and flat area percentage aspects. However, the computational cost of NC is about three times as much as SAVD. As a result, among these four methods, SAVD is the most robust and fastest.

4.2 Grid Matching

Digalakis et al.'s idea [12] of point matching is simply to match the grid points between two images. The followings are steps of this algorithm. Suppose the first image, A, in the sequence is the reference image and the next image is B.

1. $n \times n$ grids are defined overlaying on both images.

2. For a given pixel of B on the grid, find the best match using SAVD centered at various displacements around the corresponding pixel in A

(see Figure 4.29).

3. Warp image B according to the deformed grid.

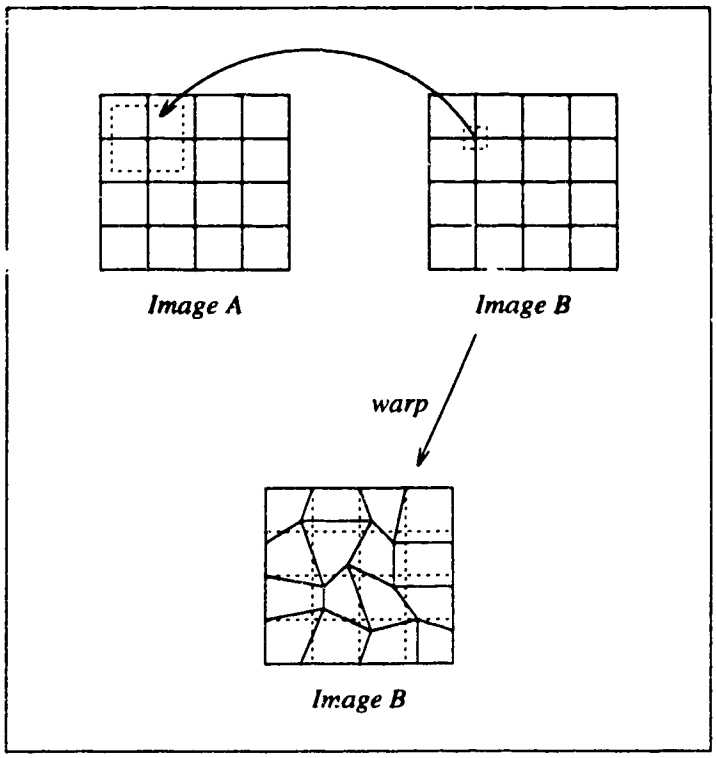
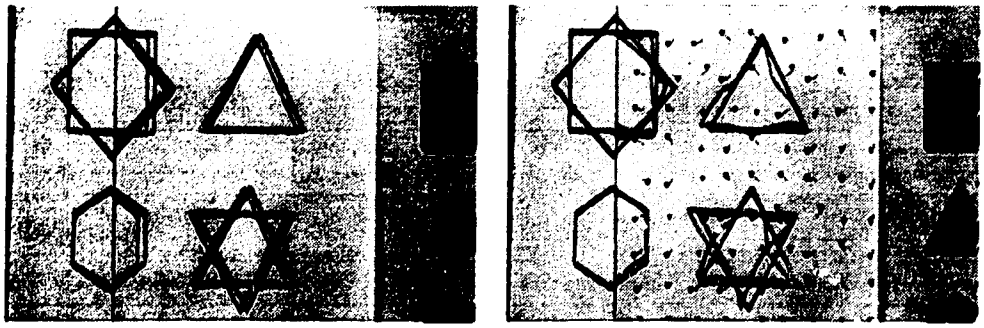


Figure 4.29: *Grid matching scheme*

The advantage of this method is that all the grid points are inherently matched which means no point matching is needed. This speeds up the overall processing time of the system. Moreover, it can handle local distortion such as imperfect lens and temperature variation of the medium which are difficult to model.

Figure 4.30(a) is an almost matched image and (b) is the result of grid matching. Note that matching only occurs in the overlap region between the two images. From the result shown in Figure 4.30(b), we see that grid



(a) *(b)*

Figure 4.30: *Grid matching*

matching performs badly. The major problem is that some grid points are located in flat areas. We know from Section 4.1.5 that none of the similarity measures work in those areas. Consequently, the random matches create and absurd matching result.

4.3 Point Matching by Relaxation

Ton and Jain [31] used a relaxation method to handle the point correspondence problem. Let $A = \{A_1, A_2, \cdots, A_m\}$ be a set of control points in image A, and $B = \{B_1, B_2, \cdots, B_n\}$ be a set of control points in image B. The basic concept of this matching algorithm is based on the following assumption: If (A_i, B_j) is a true pair, then for any other point A_h in image A, there may be a

corresponding point B_r in image B such that the distance (A_i, A_h) equals the distance (B_j, B_r). This is true if (A_h, B_r) is also a correct match. If (A_i, B_j) is a true pair, then we expect the remaining $m-1$ point pairs (A_h, B_r) to provide support to (A_i, B_j). Note that m may be different from n, and so not all points will be matched.

The algorithm starts with a $m \times n$ matrix, $\boldsymbol{P} = [p_{ij}]$, where p_{ij} represents the probability of a match between points A_i and B_j. The initial value of p_{ij} is the cross-correction value of the feature points between image A and image B. Since the result of cross-correlation represents the degree of fitness, ranged from 0 to 1, it can also be interpreted as the probability of match. By using this local information, $\boldsymbol{P}^0$ is obtained. The superscript denotes the iteration number.

The proposed matching algorithm uses both the local intensity similarity of the control points and their relative distances. The support provided by the pair (A_h, B_r) towards the pair (A_i, B_j) at iteration t is given by

$$S^t(i,j;h,r) = p^t_{hr} \cdot \min(\frac{d_{ih}}{d_{jr}}, \frac{d_{jr}}{d_{ih}})$$

where p^t_{hr} represents the probability that (A_h, B_r) is a correct match at iteration t, d_{ih} represents the Euclidean distance between A_i and A_h, and d_{jr} represents the distance between B_j and B_r. Note that $0 \leq S(i,j;h,r) \leq 1$. The matching probability between points A_i and B_j at the $(t+1)$th iteration is given by

$$p^{t+1}_{ij} = p^t_{ij} \cdot S^t_{ij}$$

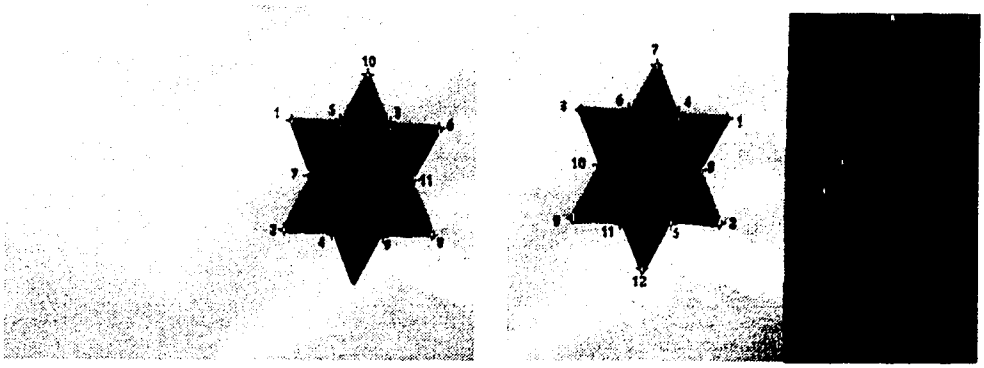
where p^t_{ij} represents the probability that (A_i, B_j) is a correct match at iteration t and S^t_{ij} represents the support of other pairs to the point pair (A_i, B_j)

at the tth iteration:

$$S_{ij}^{t} = \frac{1}{m-1} \sum_{\substack{h=1 \\ h \neq i}}^{m} [\max_{\substack{1 \leq r \leq n \\ r \neq j}} S^{t}(i,j;h,r)]. \tag{4.3}$$

The final result of $\boldsymbol{P}$ should have: 1) If (A_i, B_j) is a true pair, then p_{ij} is close to 1, otherwise it is close to 0. 2) Each row or column of $\boldsymbol{P}$ should have at most one entry close to 1. If A_i does not match to any point in B, then eventually row i of the matrix $\boldsymbol{P}$ should contain all 0's.

An example is shown to illustrate how this algorithm works. Suppose we have an picture (Figure 4.31(a)) and then we pan the camera for 50 points (Figure 4.31(b)). DET is used to detect the corners and each corner is labelled in the order of their response (e.g. corner # 1 has the strongest response to DET). Table 4.7(a), (b) and (c) shows the initial probability



(a) *(b)*

Figure 4.31: *Test images and detected corners for relaxation point matching*

matrix, $\boldsymbol{P}^0$, $\boldsymbol{P}^3$ after the 3rd iteration and $\boldsymbol{P}^5$ after the 5th iteration respectively. Each table has eleven rows which correspond to eleven corners

detected in Figure 4.31(a). Twelve columns represent the twelve corners in Figure 4.31(b). From Table 4.7(c), entry that has a value greater than 0.5 in-

	1	2	3	4	5	6	7	8	9	10	11	12
1	0.307	0.338	0.955	0.119	0.788	0.465	0.149	0.696	0.030	0.536	0.115	0.183
2	0.337	0.285	0.044	0.806	0.157	0.092	0.228	0.619	0.930	0.497	0.469	0.340
3	0.455	0.147	0.196	0.982	0.341	0.226	0.335	0.374	0.786	0.407	0.624	0.697
4	0.780	0.095	0.044	0.616	0.227	0.309	0.644	0.399	0.484	0.398	0.996	0.363
5	0.078	0.780	0.496	0.154	0.636	0.996	0.306	0.476	0.096	0.483	0.288	0.530
6	0.971	0.085	0.303	0.474	0.139	0.087	0.185	0.519	0.355	0.653	0.791	0.248
7	0.675	0.702	0.539	0.475	0.447	0.455	0.002	0.797	0.520	0.988	0.421	0.128
8	0.005	0.958	0.369	0.141	0.483	0.805	0.240	0.517	0.248	0.659	0.162	0.123
9	0.065	0.457	0.822	0.314	0.970	0.650	0.581	0.428	0.099	0.416	0.272	0.348
10	0.168	0.212	0.260	0.327	0.610	0.335	0.982	0.002	0.243	0.045	0.622	0.095
11	0.517	0.532	0.684	0.410	0.475	0.471	0.001	0.980	0.623	0.829	0.389	0.066

(a)

	1	2	3	4	5	6	7	8	9	10	11	12
1	0.000	0.000	0.269	0.000	0.057	0.001	0.000	0.021	0.000	0.003	0.000	0.000
2	0.000	0.000	0.000	0.083	0.000	0.000	0.000	0.010	0.263	0.002	0.001	0.000
3	0.000	0.000	0.000	0.262	0.000	0.000	0.000	0.000	0.044	0.000	0.007	0.017
4	0.045	0.000	0.000	0.006	0.000	0.000	0.009	0.000	0.001	0.000	0.294	0.000
5	0.000	0.044	0.001	0.000	0.009	0.308	0.000	0.001	0.000	0.001	0.000	0.002
6	0.305	0.000	0.000	0.001	0.000	0.000	0.000	0.002	0.000	0.013	0.059	0.000
7	0.009	0.013	0.002	0.000	0.000	0.000	0.000	0.036	0.001	0.200	0.000	0.000
8	0.000	0.279	0.000	0.000	0.001	0.069	0.000	0.002	0.000	0.014	0.000	0.000
9	0.000	0.000	0.067	0.000	0.251	0.010	0.004	0.000	0.000	0.000	0.000	0.000
10	0.000	0.000	0.000	0.000	0.012	0.000	0.552	0.000	0.000	0.000	0.014	0.000
11	0.001	0.001	0.009	0.000	0.000	0.000	0.000	0.162	0.004	0.043	0.000	0.008

(b)

	1	2	3	4	5	6	7	8	9	10	11	12
1	0.000	0.000	0.903	0.000	0.002	0.000	0.000	0.000	0.000	0.000	0.000	0.000
2	0.000	0.000	0.000	0.008	0.000	0.000	0.000	0.000	0.823	0.000	0.000	0.000
3	0.000	0.000	0.000	0.938	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
4	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.955	0.000
5	0.000	0.000	0.000	0.000	0.000	0.961	0.000	0.000	0.000	0.000	0.000	0.000
6	0.926	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.001	0.000
7	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.001	0.000	0.927	0.000	0.000
8	0.000	0.883	0.000	0.000	0.000	0.003	0.000	0.000	0.000	0.000	0.000	0.000
9	0.000	0.000	0.004	0.000	0.870	0.000	0.000	0.000	0.000	0.000	0.000	0.000
10	0.000	0.000	0.000	0.000	0.000	0.000	1.000	0.000	0.000	0.000	0.000	0.000
11	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.704	0.000	0.003	0.000	0.010

(c)

Table 4.7: *Values of (a) P^0 (b) P^3 and (c) P^5*

dicates a match (e.g. $1 \rightarrow 6$). The result is shown graphically in Figure 4.32.

Relaxation matching has an advantage that initially the images do not need to be close to a match position. This is because it uses only the relative distance of feature points as matching evidence without any greylevel

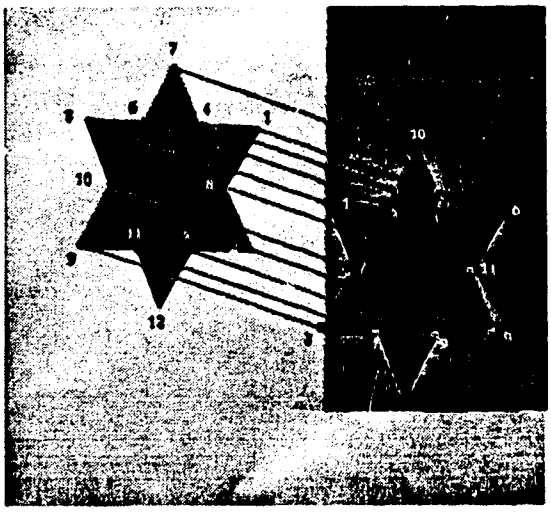


Figure 4.32: *Relaxation point matching*

similarity match. This makes it insensitive to the image intensity variation. However, on this same point, it is heavily dependent on the accuracy of the given feature position. If the corner detector has low stability, relaxation does not work at all.

4.4 Multiresolution Matching

Zheng and Chellappa [36] applied the multiresolution concept to register images. They assumed that two images are roughly matched with approximated transformation parameters such as translations (ΔX and ΔY), scale (s) and rotation (θ). Figure 4.33 is the algorithm flowchart which may help understanding how this algorithm works. First of all, both images are reduced to the lowest resolution by the factor of the power of two. Then feature points

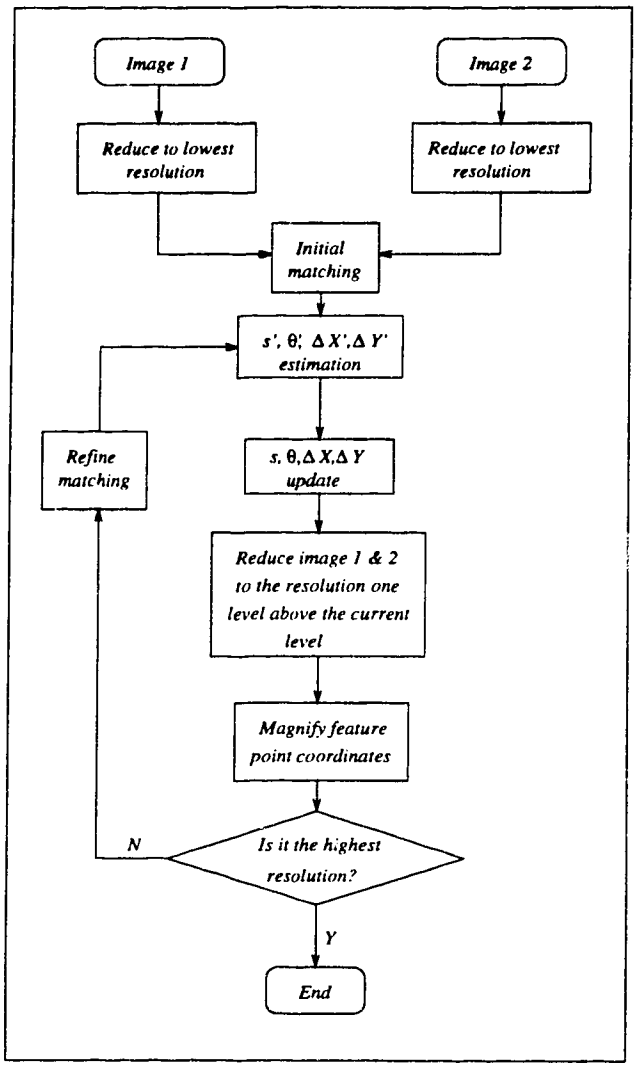


Figure 4.33: *Multiresolution matching algorithm*

are detected from each image. Since the initial transformation parameters are known, images in this resolution including the feature point coordinates are transformed. After that an initial matching takes place to find point correspondence. The matching is done by the best similarity measure found in Section 4.1.5 namely SAVD to determine the match. The template size is 17×17 and the search window size around the candidate matched point is 7×7 (see Figure 4.34). This search window size is adequate because the

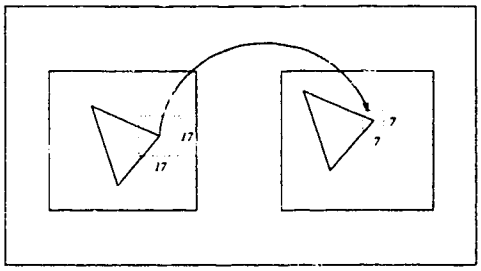


Figure 4.34: *Initial matching*

error of correspondence by DET's detector is less than three pixels. The error of correspondence means the distance between the true position and the detected position.

After the correspondences are found, those transformation parameters are estimated by a set of equations. Since Euclidean distance between the feature points only depend on the scale between the two images, and is invariant to rotation and translation, the scale factor can be estimated before the estimation of other parameters. Assume that the matched feature point pairs of $\{(X_i', Y_i') \Rightarrow (X_i^A, Y_i^A), i = 1, \cdots, N\}$ with N be the number of matched feature pairs. X_i^A and Y_i^A are the feature points coordinates of

image A. X_i' and Y_i' are the transformed feature points of image B, X_i^B and Y_i^B, using the initial $s, \theta, \Delta X$ and ΔY. Then the new scale factor is

$$\begin{bmatrix} d_1^A \\ d_2^A \\ \vdots \\ d_N^A \end{bmatrix} = s' \begin{bmatrix} d_1' \\ d_2' \\ \vdots \\ d_N' \end{bmatrix}$$

where

$$\begin{aligned} d_i' &= \sqrt{(X_i' + \overline{X_i'})^2 + (Y_i' + \overline{Y_i'})^2} \\ d_i^A &= \sqrt{(X_i^A + \overline{X_i^A})^2 + (Y_i^A + \overline{Y_i^A})^2} \\ \overline{X_i'} &= \frac{1}{N}\sum_{i=1}^{N} X_i' \\ \overline{Y_i'} &= \frac{1}{N}\sum_{i=1}^{N} Y_i' \\ \overline{X_i^A} &= \frac{1}{N}\sum_{i=1}^{N} X_i^A \\ \overline{Y_i^A} &= \frac{1}{N}\sum_{i=1}^{N} Y_i^A. \end{aligned}$$

Solving s'

$$s' = \left(\begin{bmatrix} d_1' \\ d_2' \\ \vdots \\ d_N' \end{bmatrix}^T \begin{bmatrix} d_1' \\ d_2' \\ \vdots \\ d_N' \end{bmatrix} \right)^{-1} \begin{bmatrix} d_1' \\ d_2' \\ \vdots \\ d_N' \end{bmatrix}^T \begin{bmatrix} d_1^A \\ d_2^A \\ \vdots \\ d_N^A \end{bmatrix} = \frac{\sum_{i=1}^{N} d_i' \cdot d_i^A}{\sum_{i=1}^{N} d_i' \cdot d_i'}. \qquad (4.4)$$

With the scale factor determined, the rotation and translation parameters

can be calculated.

$$\begin{bmatrix} X_i^A \\ Y_i^A \end{bmatrix} = s' \begin{bmatrix} \cos\theta' & \sin\theta' \\ -\sin\theta' & \cos\theta' \end{bmatrix} \begin{bmatrix} X_i' \\ Y_i' \end{bmatrix} + \begin{bmatrix} \Delta X' \\ \Delta Y' \end{bmatrix} \quad \text{for } i = 1, \cdots, N$$

Recall that in the beginning we assume that images are initially close to matched. Then θ' can be assumed to be small so that $\cos\theta'$ and $\sin\theta'$ can be approximated by their linear terms. As a result, the equation is

$$\begin{aligned} \begin{bmatrix} X_i^A \\ Y_i^A \end{bmatrix} &= \begin{bmatrix} 1 & \theta' \\ -\theta' & 1 \end{bmatrix} \begin{bmatrix} s'X_i' \\ s'Y_i' \end{bmatrix} + \begin{bmatrix} \Delta X' \\ \Delta Y' \end{bmatrix} \quad \text{for } i = 1, \cdots, N \\ &= \begin{bmatrix} s'X_i' + \theta' s'Y_i' \\ -\theta' s'X_i' + s'Y_i' \end{bmatrix} + \begin{bmatrix} \Delta X' \\ \Delta Y' \end{bmatrix} \\ \begin{bmatrix} X_i^A - s'X_i' \\ Y_i^A - s'Y_i' \end{bmatrix} &= \begin{bmatrix} \theta' s'Y_i' + \Delta X' \\ -\theta' s'X_i' + \Delta Y' \end{bmatrix} \\ &= \begin{bmatrix} s'Y_i' & 1 & 0 \\ s'X_i' & 0 & 1 \end{bmatrix} \begin{bmatrix} \theta' \\ \Delta X' \\ \Delta Y' \end{bmatrix} \quad \text{for } i = 1, \cdots, N. \end{aligned} \tag{4.5}$$

Then $\theta', \Delta X'$ and $\Delta Y'$ are obtained by least square method. The total transformation is obtained by combining the two transformations together:

$$\begin{aligned} \begin{bmatrix} X^A \\ Y^A \end{bmatrix} &= s' \begin{bmatrix} \cos\theta' & \sin\theta' \\ -\sin\theta' & \cos\theta' \end{bmatrix} \begin{bmatrix} X' \\ Y' \end{bmatrix} + \begin{bmatrix} \Delta X' \\ \Delta Y' \end{bmatrix} \\ &= s' \begin{bmatrix} \cos\theta' & \sin\theta' \\ -\sin\theta' & \cos\theta' \end{bmatrix} \left(s \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} X^B \\ Y^B \end{bmatrix} + \begin{bmatrix} \Delta X \\ \Delta Y \end{bmatrix} \right) + \begin{bmatrix} \Delta X' \\ \Delta Y' \end{bmatrix} \\ &= ss' \begin{bmatrix} \cos(\theta + \theta') & \sin(\theta + \theta') \\ -\sin(\theta + \theta') & \cos(\theta + \theta') \end{bmatrix} \begin{bmatrix} X^B \\ Y^B \end{bmatrix} \end{aligned}$$

$$+\begin{bmatrix} s'(\cos\theta'\Delta X + \sin\theta'\Delta Y) + \Delta X' \\ s'(-\sin\theta'\Delta X + \cos\theta'\Delta Y) + \Delta Y' \end{bmatrix}$$

As a result, the update is

$$\begin{pmatrix} \theta + \theta' \\ ss' \\ s'(\cos\theta'\Delta X + \sin\theta'\Delta Y) \\ s'(-\sin\theta'\Delta X + \cos\theta'\Delta Y) \end{pmatrix} \Rightarrow \begin{pmatrix} \theta \\ s \\ \Delta X \\ \Delta Y \end{pmatrix}. \tag{4.6}$$

Now, we have the newly updated transformation parameters at the lowest resolution level. Then feature points coordinates are magnified upward in the resolution hierarchy and image 1 & 2 at original resolution are reduced to one level higher than the current resolution level. Refine matching is done at this level but with smaller search window size this time. The new search window size is 3×3 because in each magnification. The image size is twice as big as before, and each pixel is expanded into four pixels (see Figure 4.35). Therefore, a 3×3 neighborhood is enough to cover all the possible positions in the next level.

After the new matches are found, we use Equation (4.4), (4.5) and (4.6) to estimate $(s, \theta, \Delta X, \Delta Y)$ again. We iterate the magnification, matching and parameters estimation until the original resolution is reached.

Multiresolution processing refines the errors that may occur in the initial data. It is applying the concept of rough to fine searching instead of doing the search in detail at one time. Moreover, when the image is reduced to low resolution, it is virtually using more global information at constant cost of computation. For example, if we reduce an image four times (I_2) less than

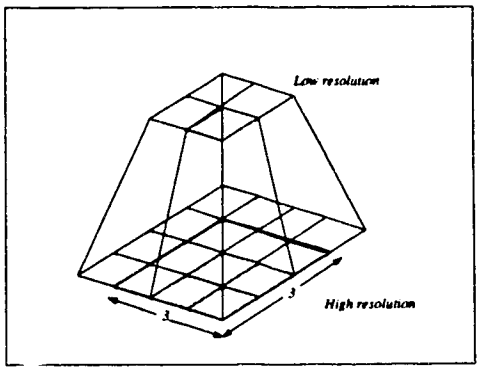


Figure 4.35: *Magnification process*

tne original size (I_1), one pixel in I_2 will represent $2^4 \times 2^4$ pixels in I_1. And if we apply a search window of 10×10 in I_2, we are actually searching over $10 \times 10 \times 2^4 \times 2^4$ pixels in I_1. As it advances upward in the resolution hierarchy, the search refines to a smaller region. It saves computation and gives more accurate result. On the other hand, the initial close to match condition must be met for it to work. It is because in the formulation of Equation (4.5), we assume that θ' is small enough to use the linear approximation. θ' must be smaller than 15° in order to have the accuracy of 3 decimal places. If the initial match is wrong, this method may not find the match.

4.5 Experimental results

Five image pairs are selected to test the ability of the point matching algorithms. Each pair focuses on one particular camera movement except the last pair which is the combination of all movement. Table 4.8 contains the correct transformation parameters between each image pair. Table 4.9, 4.10 and 4.11

contain the results obtained by grid point matching, relaxation matching and multiresolution matching respectively. In the experiment, we initially place

Image Pair	ΔX (pixel)	ΔY (pixel)	s	θ (degree)
1	160	0	1	0
2	0	−155	1	0
3	0	0	0.9	0
4	−2	6	1	−12.5
5	4	−47	1	−12.5

Table 4.8: *Correct parameters between five image pairs*

Image Pair	ΔX (pixel)	ΔY (pixel)	s	θ (degree)
1	155	1	1.005	0.280
2	2	−145	0.997	−0.278
3	1	−2	1.003	0.027
4	−1	−2	1.015	−8.973
5	5	−48	0.999	−10.979

Table 4.9: *Parameters obtained by grid point matching*

two images close to their matched position to simulate the situation when the camera position information provides erroneous data. For example, the correct horizontal translation for the first image pair is 160 pixels and we deliberately move it for 151 pixels to create a 9 pixels error. Then we apply matching algorithms to correct the error.

Grid point matching (see Table 4.9) is expected to perform badly because

Image Pair	ΔX (pixel)	ΔY (pixel)	s	θ (degree)
1	159	1	1	−0.281
2	2	−154	0.993	−0.003
3	−1	2	0.898	0.359
4	−6	4	0.992	−11.809
5	6	−50	0.994	−12.127

Table 4.10: *Parameters obtained by relaxation matching*

Image Pair	ΔX (pixel)	ΔY (pixel)	s	θ (degree)
1	160	0	1.001	0.027
2	0	−155	1.002	0.003
3	0	0	0.902	0
4	−2	5	1.001	−12.175
5	4	−47	1.002	−12.40

Table 4.11: *Parameters obtained by multiresolution matching*

- it is a local registration method in which grid points may not have a united displacement and direction. This makes the least square estimation fail.

- when some grid points fall onto flat areas, random registered points result. This causes the random movement in Figure 4.30.

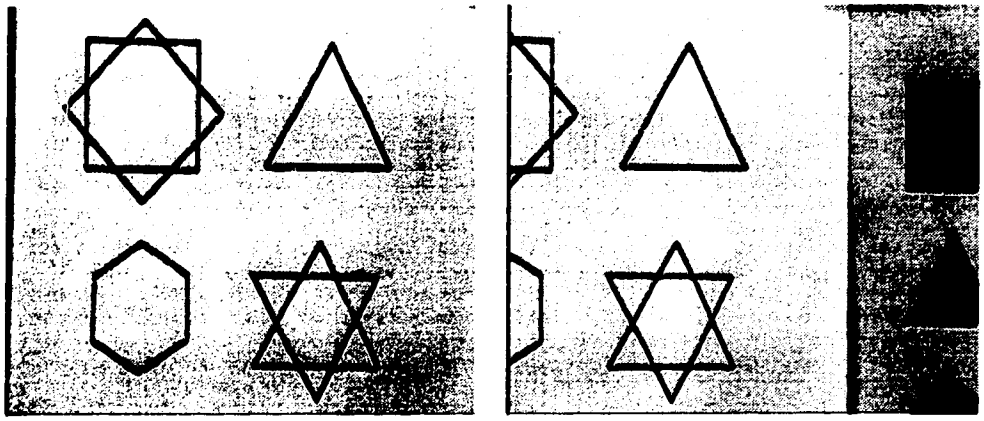
On the other hand, relaxation matching provides more promising results. In Table 4.10, we can notice that this method can estimate the movement parameters within our tolerance (i.e. ± 1 pixel and $\pm 1°$). However, it fails to locate the position of the registration when rotation and scaling is involved. This is because it cannot refine the position of feature points provided by the corner detector. For example if a true match occurs at (16,16) in I_1 and (20,20) in I_2, but the feature points found by corner detector are (16,16) in I_1 and (20,22) in I_2. Although relaxation can tolerate the error and claims that as a match, the least square estimation (Equation 4.5) are sensitive to this error. This happens in image pairs 3, 4 and 5. All the matched point pairs are two to three pixels off the true match positions.

Finally, we found that multiresolution matching is the most robust algorithm. In Table 4.11, it gives exact position in translations, even when rotation and scaling occur. This is because it uses multiresolution refinement matching. Its initial matching process can adjust the errors created by the corner detector. And as advancing up each resolution level, the match is refined. Figure 4.36, 4.37, 4.38 and 4.39 are the resulting images. Figure 4.40, 4.41 are the results of real scene matching. One should notice that multiresolution matching method corrects the errors caused by the camera's

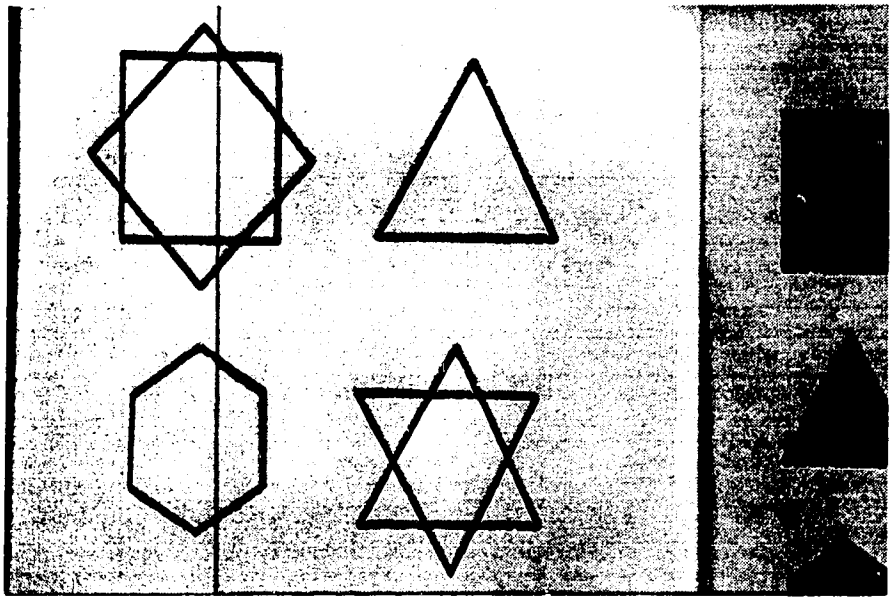
initial position information.

I have tried to use the relaxation method as the initial matching module in the multiresolution matching. I wanted to use the advantage of relaxation which can match points without being initially close to match position. However, relaxation fails to find unique match in lower resolution. As a result, I still need to supply initial position data to the algorithm.

In summary, grid point matching is not adequate in our application because of its failure in matching grid point in flat areas. Relaxation is susceptible to the instability of the corner detector. Although it can recover the transformation parameters in each isolated case, it fails to find the exact location in the mixed situation. Multiresolution matching is found to give the best results because it can refine the initial matching through out its multiresolution hierarchy and it also makes use of global information of the image to enhance the matching quality.

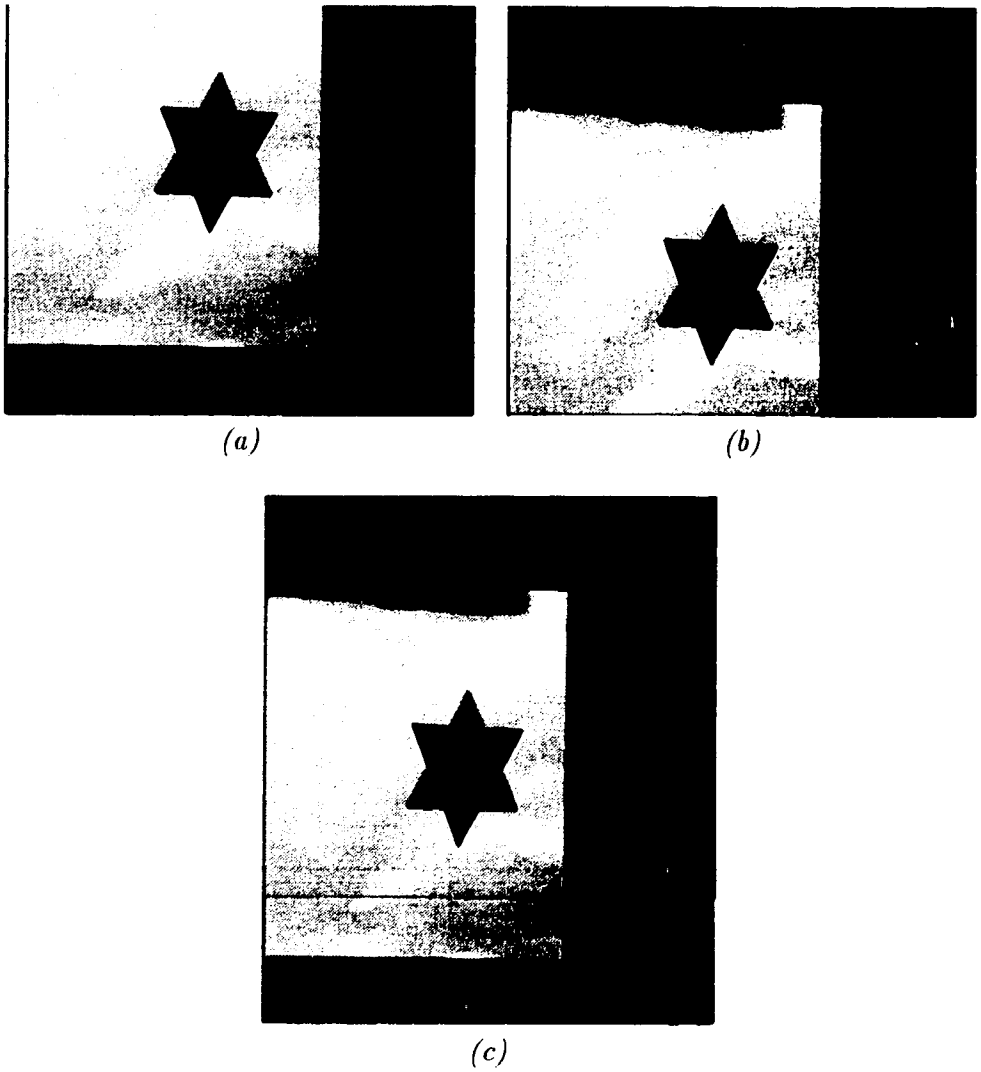


(a) *(b)*



(c)

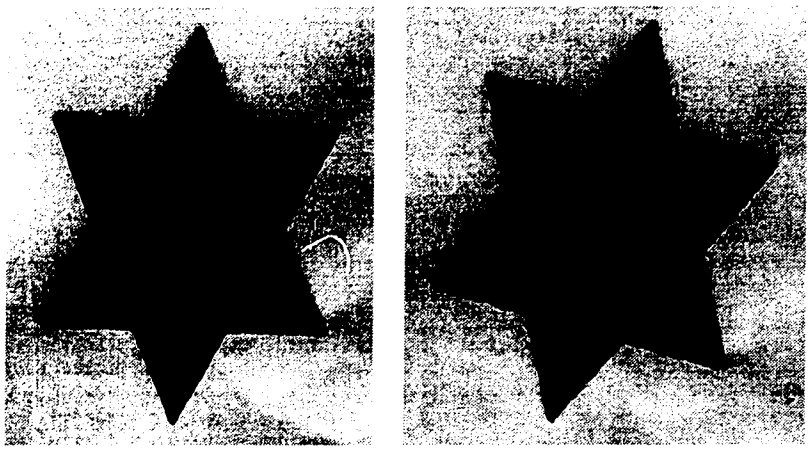
Figure 4.36: $\Delta X = 160$, $\Delta Y = 0$, $s = 1.001$, $\theta = 0.027°$ *by Multiresolution Matching*



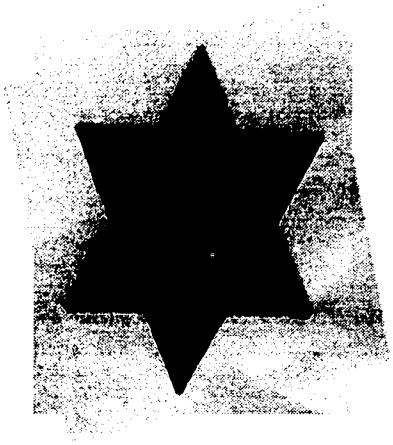
(a) *(b)*

(c)

Figure 4.37: $\Delta X = 0$, $\Delta Y = -155$, $s = 1.002$, $\theta = 0.084°$ *by Multiresolution Matching*

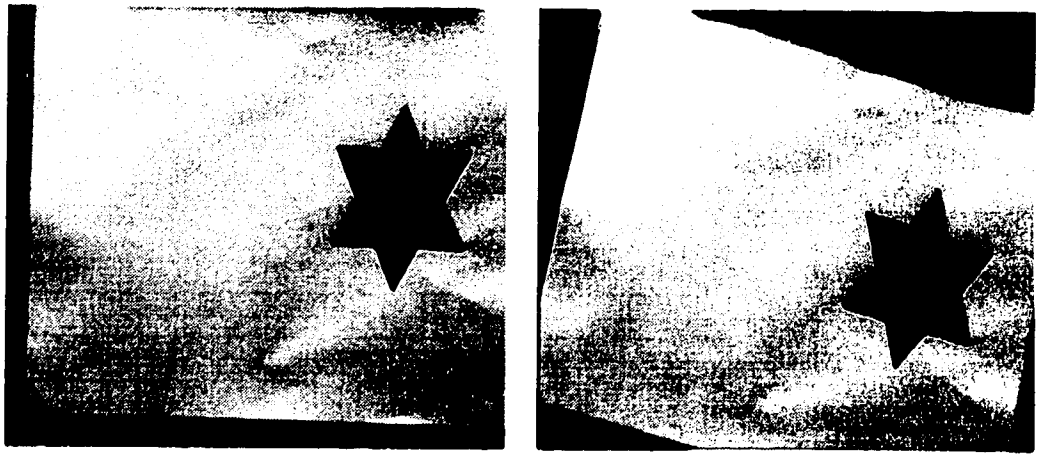


(a) *(b)*

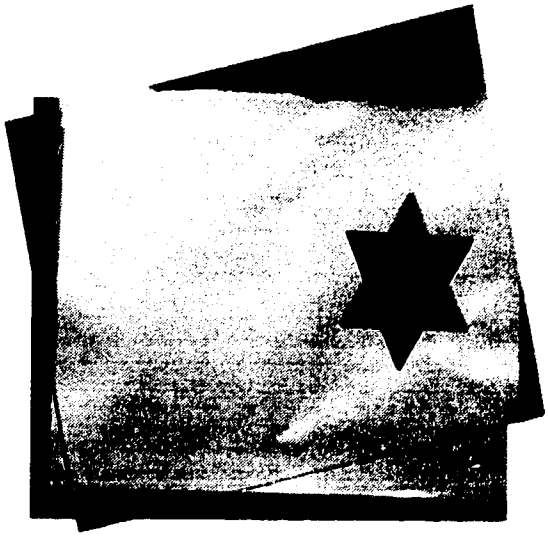


(c)

Figure 4.38: $\Delta X = -2$, $\Delta Y = 5$, $s = 1.001$, $\theta = 12.175°$ *by Multiresolution Matching*



(a) *(b)*



(c)

Figure 4.39: $\Delta X = 4$, $\Delta Y = -47$, $s = 1.002$, $\theta = 12.40°$ *by Multiresolution Matching*

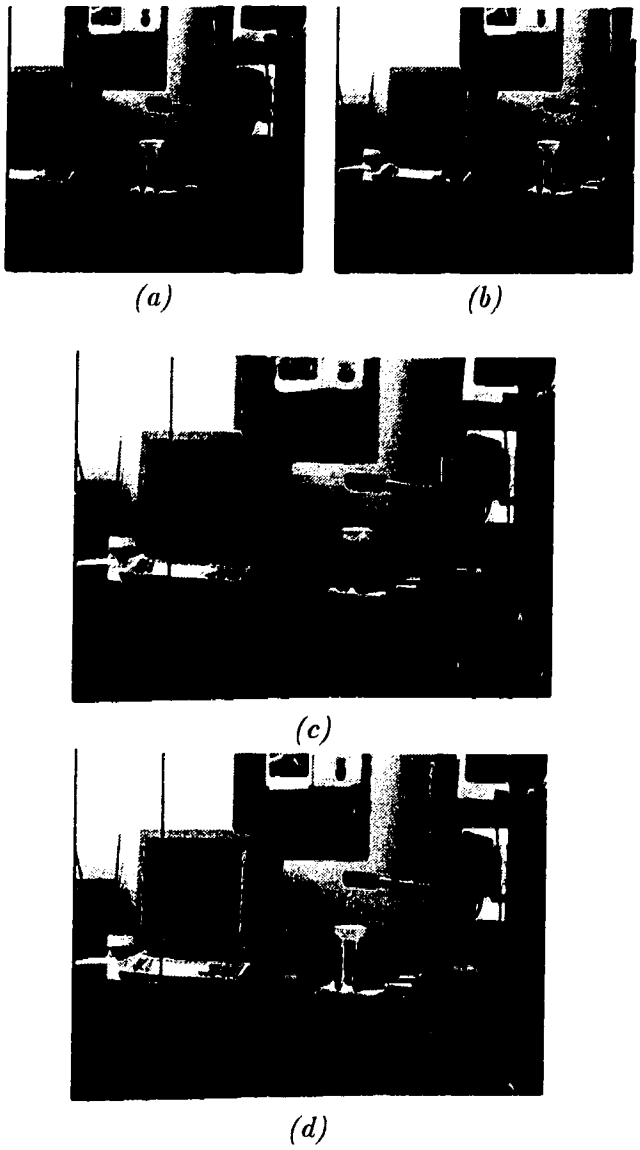


Figure 4.40: *(a), (b) Grabbed images, (c) initial position mapping with* $\beta = 6.65°$, *(d) refine match by multiresolution method with* $\Delta X = -107$, $\Delta Y = 2$, $s = 1.001$, $\theta = 0.023°$

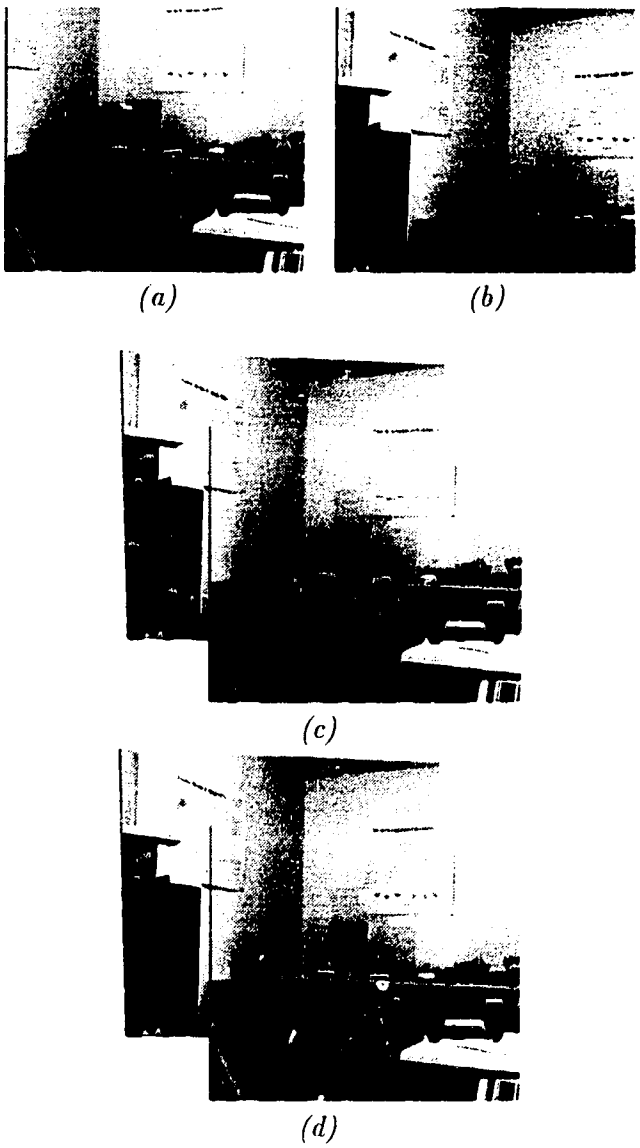


Figure 4.41: *(a), (b) Grabbed images, (c) initial position mapping with* $\beta = -6.28°$ *and* $\gamma = -6.20°$*, (d) refine match by multiresolution method with* $\Delta X = -133$, $\Delta Y = -114$, $s = 1.002$, $\theta = 0.003°$

Chapter 5

Conclusion

An automatic image registration system is built with two major components: position mapping module and the feature-based matching module. Each component was tested and the results demonstrated the validity and robustness of the algorithm applied.

The mathematical model of position mapping is shown to be correct given accurate positional information. This technique exploits information about the camera motion in order to derive a mapping function relating pixels in two successive images, excluding certain peripheral regions. Since mapping pixels between two frames is costly in terms of computations needed, we instead compute the coordinates of the new frame in the original frame of reference and then use a warping algorithm to map pixels. As a result, registration of images becomes not only easy but also efficient. Moreover, this technique has the advantages of being insensitive to grey level modifications, and highly accurate in registering images that are offset by rotation and/or translation.

However, this method is sensitive to inaccurate sensor readings and therefore fine tuning of registered images is required. This is one reason for exploring feature-based matching to correct the possible error.

Our feature-based matching is a versatile algorithm which can endure errors from the position mapping module and gives an accurate match. Beaudet's DET is chosen to be the feature detector because it has high stability, is fast and is able to recognize most obvious corners. DET can detect corners on either solid or line objects in which corners are considered hard to detect by other existing methods. We applied a multiresolution technique to match the feature points generated by DET. It uses multiresolution refinement technique to minimize the initial errors and also converts from global to local search to reduce the computation. Experiments show that it can handle all possible image displacement and orientation errors such as translation, rotation and scaling. As a result, it is the best the correctional process for the position mapping module.

Finally, this system has great potential in many practical applications, because there is no restriction on how the camera should be oriented in capturing images. The feature-based matching can correct errors from the positioning device. This system is ideal in applications such as underwater inspection and satellite imaging where the camera cannot always be looking perpendicularly at the scene.

5.1 Future Work

There are many improvements can be made in this system:

1. Detail depth data
2. Accurate warping algorithm
3. General feature detection method
4. General initial matching method
5. Nonlinear estimation of 3-D movement

A more detailed depth information of the scene is essential to the quality in matching a complex 3-D scene. Although our mathematical model is suitable for mapping 3-D object onto a 2-D image, the depth data is always needed. In all of our experiments, we must manually estimate the depth of a flat object and approximate the average depth when dealing with a real scene. Ideally each pixel in the image should have its own depth but because of the financial limitation for this research, approximation is used. The effects of discontinuous depth can be seen in Figure 4.40(d) where the white cord at the back of the camera cannot be matched because its distance from our viewing camera is different from the PEPSI cup and the monitor.

Another improvement is closely related to the depth problem which is the quality of the warping algorithm. We use a pair of linear polynomial functions (Equation (2.7) and (2.8)) to map the four corners of the initial image onto the perspective shape but the mapping of the content pixels obey the linear polynomial instead of the mathematical model (Equation (2.5) and (2.6)). Although we have the correct four corners, the content of the image can be wrongly mapped. This effect occurs when the objects in the scene have great difference in depth. We cannot use the original mapping functions (Equation

(2.5) and (2.6)) because it is a forward mapping function but we need a backward mapping function for the warping. Therefore, a better function that can perform closer to the inverse of the original mapping function is needed.

In our feature-based matching algorithm, corner is the prime feature. But in a real world situation, corners may not exist. As a result, we need to have a more general detector which can find corners and another kinds of feature such as line end. One possible method is the wavelet-transform feature detection. Zheng [36] used wavelet transform to detect features on a satellite ground image and their results are extraordinary.

In the experiments in Section 4.5, we need to initially place the images close to the match position. If the initial matching module can find match points without this condition, this algorithm can be generalized even with no position mapping in the beginning. Moreover, this matching method should be robust enough in various image resolutions.

Finally, as far as this research goes, we did not discover any mathematical formula that can estimate the camera motion parameters, $\alpha, \beta, \gamma, \Delta X, \Delta Y$ and ΔZ, in Equation (2.5) and (2.6). Theoretically, we need to write Equation (2.5) and (2.6) into the form of linear system of equations and use nonlinear least square to obtain the parameters. This can be done in the future.

Bibliography

[1] R. Bajcsy and S. Kovačič. Multiresoultion elastic matching. *Computer Vision, Graphics, and Image Processing*, 46:1–21, 1989.

[2] D. I. Barnea and H. F. Silverman. A class of algorithms for fast digital registration. *IEEE Transactions on Computer*, 21:179–186, 1972.

[3] P. R. Beaudet. Rotationally invariant image operators. In *Proceedings of International Joint Conference on Pattern Recognition*, pages 579–583, 1978.

[4] Lisa G. Brown. A survey of image registration techniques. *ACM Computing Surveys*, 24(4):325–376, December 1992.

[5] D. J. Burr. A dynamic model for image registration. *Computer Graphics and Image Processing*, 15:102–112, 1981.

[6] P. J. Burt, C. Yen, and X. Xu. Local correlation measures for motion analysis — a comparative study. In *Proceedings of IEEE International Conference on Pattern Recognition and Image Processing*, pages 269–274, 1982.

[7] E. De Castro and C. Morandi. Registration of translated and rotated images using finite Fourier Transforms. *IEEE Transactons on Pattern Analysis and Machine Intelligence*, 9(5):700–703, September 1987.

[8] L-H Chen. A two-phase area-level line/edge detector. *Pattern Recognition*, 25(1):55–63, 1992.

[9] J. K. Cheng and T. S. Huang. Image registration by matching relational structures. *Pattern Recognition*, 17(1):149–159, 1984.

[10] J. Cooper, S. Venkatesh, and L. Kitchen. Early jump-out corner detectors. *IEEE Transactons on Pattern Analysis and Machine Intelligence*, 15(8):823–828, August 1993.

[11] R. Deriche and G. Giraudon. Accurate corner detection: An analytical study. In *Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 66–70, 1990.

[12] V. V. Digalakis, D. G. Manolakis, P. Lazaridis, and V. K. Ingle. Enhancement of digital angiographic images with misregistration correction and spatially-adaptive matched filtering. In *Proceedings of SPIE – The International Society for Optical Engineering*, volume 1818, pages 1264–1270, 1992.

[13] J. Flusser. An adaptive method for image registration. *Pattern Recognition*, 25(1):45–54, 1992.

[14] W. Förstner. A feature based correspondence algorithm for image matching. In *Proceedings of International Society of Photogrammetry*

and Remote Sensing ISPRS, Symposium Comm. III, volume 19, pages 150–166, 1986.

[15] W. Förstner and E. Gülch. A fast operator for detection and precise location of distinct points, corners and centres of circular features. In *Proceedings of ISPRS Intercommission Workshop*, pages 281–305, 1986.

[16] M. E. Fuller and M. Ehlers. Automated system for image co-registration using interest clump matching. *Photogrammetry and Primary Data Acquisition Technical Paper 91 ACSM ASPRS Annual Convention*, 5:93–102, 1991.

[17] A. Goshtasby. Image registration by local approximation methods. *Image and Vision Computing*, 6(4):255–261, November 1988.

[18] A. Goshtasby. Registration of images with geometric distortions. *IEEE Transactions on Geoscience and Remote Sensing*, 26(1):60–64, January 1988.

[19] M. Herbin, A. Venot, J. Y. Devaux, E. Walterand J. F. Lebruchec, L. Dubertret, and J. C. Roucayrol. Automated registration of dissimilar images: Application to medical imagery. *Computer Vision, Graphics, and Image Processing*, 47:77–88, 1989.

[20] L. Kitchen and A. Rosenfeld. Gray-level corner detection. *Pattern Recognition Letters*, 1:95–102, December 1982.

[21] C. R. Kube. A review of underwater imaging techniques. Computing Science Department, University of Alberta, May 1993.

[22] C. D. Kuglin and D. C. Hines. The phase correlation image alignment method. In *Proceedings of IEEE International Conference on Cybernetics and Society*, pages 163–165, 1975.

[23] Y. Lü. Interest operator and fast implementation. *International Archives of Photogrammetry and Remote Sensing*, 27(3):491–500, 1988.

[24] M. Merickel. 3D reconstruction: The registration problem. *Computer Vision, Graphics, and Image Processing*, 42(2):206–219, 1988.

[25] E. M. Mikhail and F. Ackermann. *Observations and Least Squares*, chapter 4, pages 76–79. IEP, New York, 1976.

[26] A. A. Mustafa and M. A. Ganter. Efficient image registration by intensity combinatorial minimization. In *Proceedings of Vision Interface '94, Canadian Image Processing and Pattern Recognition Society*, pages 163–171, 1994.

[27] K. Rangarajan, M. Shah, and D. V. Brackle. Optimal corner detector. In *Proceedings of IEEE International Conference on Computer Vision*, pages 90–94, 1988.

[28] A. Rosenfeld and A. C. Kak. *Digital Picture Processing*, volume 1 and 2. Academic Press Inc. Ltd., London, 2nd edition, 1982.

[29] M. Svedlow, C. D. McGillem, and P. E. Anuta. Experimental examination of similarity measures and preprocessing methods used for image registration. In *The Symposium on Machine Processing of Remotely Sensed Data*, pages 4A–9, 1976.

[30] D. Terzopoulos, A. Witkin, and M. Kass. Energy constraints on deformable models: Recovering shape and non-rigid motion. In *Proceedings of AAAI 87 (July)*, volume 2, pages 755–760, 1987.

[31] J. Ton and A. K. Jain. Registering landsat images by point matching. *IEEE Transactions on Geoscience and Remote Sensing*, 27(5):642–651, September 1989.

[32] P. A. van den Elsen, E. D. Pol, and M. A. Viergever. Medical image matching - a review with classification. *IEEE Engineering in Medicine and Biology*, 12(1):26–39, March 1993.

[33] A. Venot, J. F. Lebruchec, and J. C. Roucayrol. A new class of similarity measures for robust image registration. *Computer Vision, Graphics and Image Processing*, 28:176–184, 1984.

[34] L. Wang and T. Pavlidis. Direct gray-scale extraction of features for character recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(10):1053–1067, October 1993.

[35] X. Xie, R. Sudhakar, and H. Zhuang. Corner detection by a cost minimization approach. *Pattern Recognition*, 26(8):1235–1243, 1993.

[36] Q. Zheng and R. Chellappa. A computational vision approach to image registration. *IEEE Transactions on Image Processing*, 2(3):311–326, July 1993.

[37] J. Zong, J. C. Li, and A. F. Schenk Application of Förstner interest operator in automatic orientation system. *Photogrammetry and Primary*

Data Acquisition Technical Paper 91 ACSM ASPRS Annual Convention, 5:440–448, 1991.

[38] O. A. Zuniga and R. M. Haralick. Corner detection using the facet model. In *Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 30–37, 1983.

Appendix A

Program code

This appendix contains most of the code of the automatic registration system. They are all written in C and can be compiled using Sun's C compiler or the Gnu C compiler. Figure A.1 shows the file relation of the registration system. They are grouped by function. This appendix contains only the representative code. For example, only the code of Beaudet's DET, corner4.c, is listed here. The source code can be found in

`/usr/samson-pk/prof/tony/Collins/src`

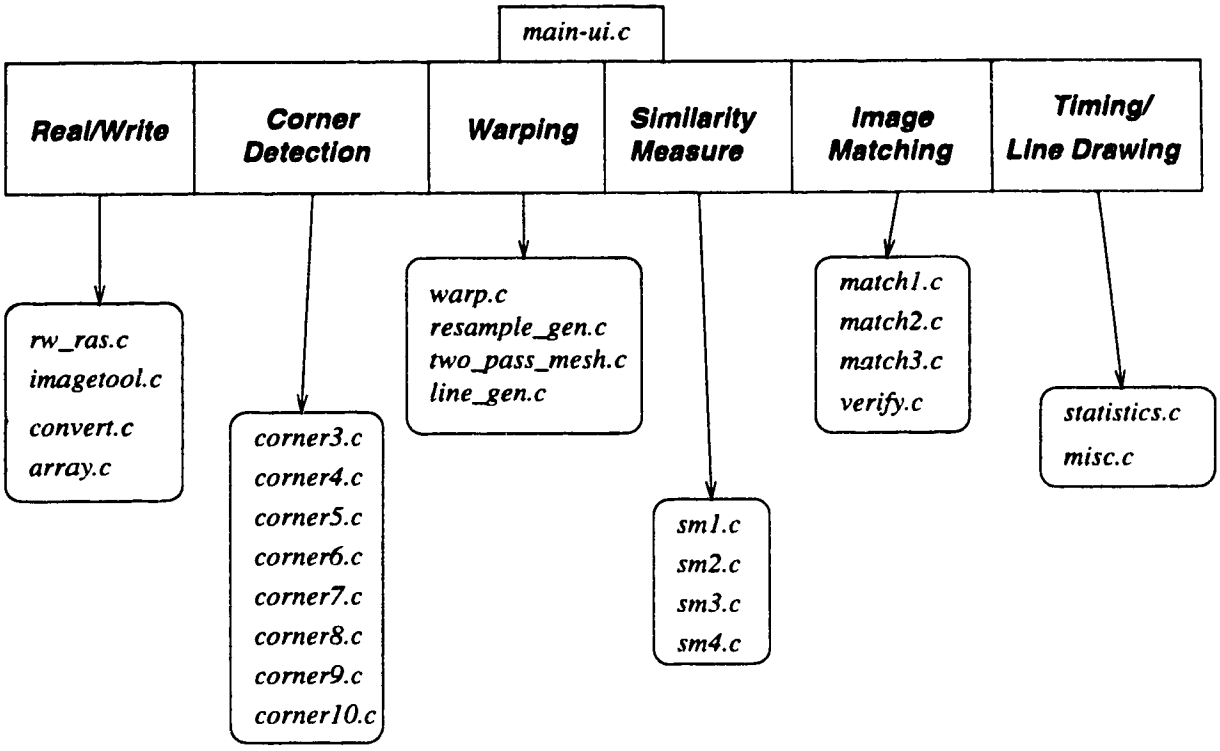


Figure A.1: *File relation of the registration system*

```
#include <stdio.h>
#include <sys/file.h>
#include <rasterfile.h>
#include "../include/data_struct.h"
#include "../include/constants.h"
#include "../include/struct.h"

extern IMAGE    im1, im2, imf;
extern int      file_flag;
extern int      time2_flag;

/*************************************************************
 * Module Summary : This group of functions are responsible  *
 *                  of reading and writting raster files.    *
 *                                                           *
 * Functions : read_im, reads an image in raster file format.*
 *             gsize, reads the size of an image in raster   *
 *                    file format.                           *
 *             get_size, reads the size of a group of images *
 *                    in raster file format and obtains the  *
 *                    maximum width and height of them.      *
 *             write_im, writes a raster file.               *
 *                                                           *
 * Author : Sergio Licardie                                  *
 * Date :   Jan. 8, 1993                                     *
 *                                                           *
 * Modifications Summary                                     *
 *                                                           *
 * Modified by Collins Chien, June 17 1994.                  *
 *                                                           *
 *************************************************************/

/*************************************************************
 *                                                           *
 *   Function name : read_im                                 *
 *                                                           *
 *   Description : Take an image in raster file format from  *
 *                 and stores it into a previously allocated *
 *                 memory vector.                            *
 *                 The memory vector can be larger or smaller*
 *                 than the raster to load. If larger, an    *
 *                 offset can be specified in order to load  *
 *                 that image in any place of the memory     *
 *                 vector. If shorter, only the top left     *
 *                 corner that fits will be loaded into the  *
 *                 memory vector.                            *
 *                 The memory vector can be of 3 different   *
 *                 types, which are specified by the para-   *
 *                 meter vector_type. The 3 types are the    *
 *                 following: Integer, Double, and Complex.  *
 *                 The pointer is properly casted in order   *
 *                 to cover the different types.             *
 *                                                           *
 *   Parameters  : char *fname - the name of the raster file *
 *                               to be read.                 *
 *                 int  *raster - memory vector where to put *
 *                                the raster image.          *
 *                 int width - width of the memory vector.   *
 *                 int height - height of the memory vector. *
 *                 int x_off - offset from left to load the  *
 *                             raster.                       *
 *                 int y_off - offset from top to load the   *
 *                             raster.                       *
 *                 int vector _type - type of memory vector  *
 *                             in which the image will be    *
 *                             loaded. 0 -> Unsigned char.   *
 *                                     1 -> Short.           *
 *                                     2 -> Integer          *
 *                                     3 -> Float.           *
 *                                     4 -> Double.          *
 *                                     5 -> Complex          *
 *                                                           *
 *   Outputs     : none.                                     *
 *                                                           *
 *   Returned Value : SUCCESS operation was succesfull.      *
 *                    FAIL    operation failed.              *
 *                    The space to which the raster pointer  *
 *                    points to is loaded with the image.    *
 *                                                           *
 *   Related functions : none.                               *
 * ----------------------------------------------------------*
 *                                                           *
 *   Original : Bob Gregorish.                               *
 *   Modified : Sergio Licardie, 23/07/91                    *
 *              Modified to own needs from im2int.           *
 *              Collins Chien, 17/06/94                      *
 *              Able to read raster image properly           *
 *                                                           *
 *************************************************************/

read_im(fname,raster,width,height,x_off,y_off,vector_type)

char *fname;
int  *raster;
int width, height, x_off, y_off, vector_type;

{

FILE *infile;
register int i,j,idx;
struct rasterfile image;
int greylevel;

int w,h;
int odd;
char c;

unsigned char *chr_p;
short  *shr_p;
float  *flt_p;
double *dou_p;
struct complex_im *com_p;

/* open the image file */

if ((infile = fopen (fname, "r")) == NULL)
```

```
{
   fprintf(stderr,"can't open %s\n",fname);
   return(FAIL);
}

/* read file header */

if (fread (&image, sizeof(image), 1, infile) != 1)
{
   fprintf(stderr,"error reading discriptor\n");
   return(FAIL);
}

switch(file_flag) {
        case 1: im1.ras_maplength = image.ras_maplength;
                break;
        case 2: im2.ras_maplength = image.ras_maplength;
                break;
        case 3: imf.ras_maplength = image.ras_maplength;
                break;
}

/* Flush the color table */

for (i = 0; i < image.ras_maplength; i++) {
   if (getc(infile) == EOF)
   {
      fprintf (stderr, "unexpected EOF 1\n");
      return (FAIL);
   }
}

w = image.ras_width;
h = image.ras_height;
odd = w % 2;

fprintf(stderr,"reading image %s , width = %d, height = %d\n",fname,w,h);

/* read the image grey values from the image raster file, convert to
 * the proper type according to the vector_type variable and store in
 * the vector in the appropiate format. (default integer)
 */
for (i = 0; i < h; i++)
{
   if (i+y_off < height)
   {
      for (j = 0; j < w; j++)
      {
         if (j+x_off < width)
         {
            greylevel = (int)(getc(infile));
            idx       = (i+y_off)*width+j+x_off;
            if (vector_type == DOU_IM)
            {
               dou_p = (double *)raster;
               dou_p[idx] = (double)greylevel;
            }
            else if (vector_type == COM_IM)
            {
               com_p = (struct complex_im *)raster;
               com_p[idx].re = (double)greylevel;
               com_p[idx].im = 0.0;
            }
            else if (vector_type == FLT_IM)
            {
               flt_p = (float *)raster;
               flt_p[idx] = (float)greylevel;
            }
            else if (vector_type == SHR_IM)
            {
               shr_p = (short *)raster;
               shr_p[idx] = (short)greylevel;
            }
            else if (vector_type == CHR_IM)
            {
               chr_p = (unsigned char *)raster;
               chr_p[idx] = (unsigned char)greylevel;
            }
            else {
                if( time2_flag == 1) {
                  raster[idx] = 2*greylevel;
                  if(raster[idx] > 255) raster[idx] = 255;
                } else
                 raster[idx] = greylevel;
            }
         }
      }
      if(odd)
        getc(infile);
   }
}

(void) fclose(infile);
return(SUCCESS);

}

/*************************************************************
 *                                                           *
 *   Function name : write_im                                *
 *                                                           *
 *   Description : Take an image memory vector and stores it *
 *                 in a file using raster format. The vector *
 *                 be saved complete or only a part of it.   *
 *                 If only a section of it is to be saved,   *
 *                 the origin of such section will be defined*
 *                 by the the variables ox and oy, and the   *
 *                 size by *width and height. In the case in *
 *                 which the whole image is to be saved, the *
 *                 parameters ox = 0, oy = 0, width =&t_width*
 *                 and height = &t_height.                   *
 *                 The parameter *width is a pointer because *
 *                 in the routine, the width of the image in *
 *                 raster file is validated to be a multiple *
 *                 of 16.                                    *
 *                                                           *
```

```
 *   Parameters  : char *fname - the name of the raster file *
 *                              to be created.               *
 *                 int *raster - pointer to the memory vector*
 *                              where the image is stored.   *
 *                 int t_width - total width of the vector.  *
 *                 int t_height - total height of the vector.*
 *                 int *width - width of the part of the     *
 *                              vector to be saved.          *
 *                 int height - height of the part of the    *
 *                              vector to be saved.          *
 *                 int ox,oy  - Top left origin of the vector*
 *                              to start saving.             *
 *                 int vector _type - type of memory vector  *
 *                             in which the image is stored. *
 *                                    0 -> Integer.          *
 *                                    1 -> Short.            *
 *                                    2 -> Integer           *
 *                                    3 -> Float.            *
 *                                    4 -> Double.           *
 *                                    5 -> Complex           *
 *                                                           *
 *   Outputs     : none.                                     *
 *                                                           *
 *   Returned Value : SUCCESS operation was succesfull.      *
 *                    FAIL    operation failed.              *
 *                    The image stored in the memory vector  *
 *                    pointed by raster is dumped into a     *
 *                    file with raster format.               *
 *                                                           *
 *   Related functions : none.                               *
 * ----------------------------------------------------------*
 *                                                           *
 *   Original : Bob Gregorish.                               *
 *   Modified : Sergio Licardie, 23/07/91                    *
 *              Modified to own needs from im2int.           *
 *              Collins Chien, 17/06/94                      *
 *              Able to write raster file properly           *
 *                                                           *
 ************************************************************/

write_im(fname,raster,t_width,t_height,width,height,ox,oy,vector_type)
char  *fname;
int  *raster;
int t_width,t_height,*width,height,ox,oy,vector_type;
{
FILE *outfile;
register int i,j,idx;
struct rasterfile image;
float width_f;
int   width_i;

unsigned char val;
unsigned char *chr_p;
short  *shr_p;
float  *flt_p;
double *dou_p;
struct complex_im *com_p;
unsigned char c;
int odd;


/* create and initialize the file header */

   odd = t_width % 2;
   image.ras_magic = RAS_MAGIC;
   image.ras_width = t_width;
   image.ras_height = height;
   image.ras_depth = 8;
   image.ras_length = (t_width+odd)*height;
   image.ras_type = RT_STANDARD;
   image.ras_maptype = RMT_NONE;
   image.ras_maplength = 0;
/* open the image file to which data is to be written */

   if ((outfile = fopen(fname, "w")) == NULL)
   {
      fprintf (stderr, "can't open %s\n", fname);
      return(FAIL);
   }

/* write the header to the file */

   if (fwrite (&image, sizeof(image), 1, outfile) != 1)
   {
      fprintf(stderr,"error in writing file header\n");
      return(FAIL);
   }
/* convert the values in the input memory vector to
 * unsigned characters and write them to the image file.
 * Taking into account the origin x,y position in the original
 * raster so we can cut a width*height piece from it.
 */

   printf("saving image %s, width = %d, height = %d\n",fname,t_width,height);
   c = (unsigned char)raster[0];
   for (j = 0; j < height; j++) {
     for(i=0; i < t_width; i++) {
        val = (unsigned char)raster[j*t_width+i];
        putc(val, outfile);
     }
     if(odd)
        putc(c,outfile);
   }

/* close the file */

   (void) fclose(outfile);
   return(SUCCESS);
}


/************************************************************
 *                                                          *
 *   Function name : get_size                               *
 *                                                          *
 *   Description : This function reads the headers of several*
```

```
 *                various images, stores their parameters in*
 *                a structure and determines the maximum    *
 *                width and height presente among those     *
 *                raster files.                             *
 *                                                          *
 *   Parameters  : int nimage - number of images to be read. *
 *                 int *max_width - maximum width found among*
 *                                  the images.             *
 *                 int *max_height-maximum height found among*
 *                                  the images.             *
 *             struct im_file *im_f-structure vector in which*
 *                                  the parameters of each image *
 *                                  will be stored.         *
 *                                                          *
 *   Outputs     : none.                                    *
 *                                                          *
 *   Returned Value : SUCCESS operation was succesfull.     *
 *                    FAIL    operation failed.             *
 *                                                          *
 *   Related functions : gsize.                             *
 * ---------------------------------------------------------*
 *                                                          *
 *   Original : Sergio Licardie, 12/06/92                   *
 *   Modified :                                             *
 *                                                          *
 ************************************************************/

get_size(nimage,max_width,max_height,im_f)

int  nimage;
int  *max_width, *max_height;
struct im_file *im_f;

{

FILE   *infile;
struct rasterfile image;
int    i,width,height,sts;

*max_width = *max_height;

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

   sts = g_size(im_f[i].name,&width,&height);

   if (sts == FAIL)
      return(FAIL);

   im_f[i].wi = width;
   im_f[i].he = height;
   width    += 10;
   height   += 10;
   if (height > *max_height)
      *max_height   = height;
   if (width > *max_width)
      *max_width    = width;
}

return(SUCCESS);

}

/************************************************************
 *                                                          *
 *   Function name : g_size                                 *
 *                                                          *
 *   Description : This function reads the header of an     *
 *                 image, and returns its width and height  *
 *                 the paramters provided                   *
 *                                                          *
 *   Parameters  : char *name - name of the raster file.    *
 *                 int *width - used to return the width of *
 *                              the image.                  *
 *                 int *height- used to return the height of *
 *                              the image.                  *
 *                                                          *
 *   Outputs     : none.                                    *
 *                                                          *
 *   Returned Value : SUCCESS operation was succesfull.     *
 *                    FAIL    operation failed.             *
 *                    Returns the width and the height of   *
 *                    image read in *width and *height.     *
 *                                                          *
 *   Related functions : none.                              *
 * ---------------------------------------------------------*
 *                                                          *
 *   Original : Sergio Licardie, 12/06/92                   *
 *   Modified :                                             *
 *                                                          *
 ************************************************************/

g_size(name,width,height)

char *name;
int  *width, *height;

{

FILE   *infile;
struct rasterfile image;

*width = *height = 0;

/* open the image file */

if ((infile = fopen (name, "r")) == NULL)
{
   fprintf(stderr,"can't open %s\n",name);
   return(FAIL);
}

/* read file header */

if (fread (&image, sizeof(image), 1, infile) != 1)
```

```
{
   fprintf(stderr,"error reading discriptor\n");
   return(FAIL);
}

*width  = image.ras_width;
*height = image.ras_height;

fprintf(stderr,"reading image header %s , width = %d, height = %d\n",name,*width
,*height);

return(SUCCESS);

}
```

```
#include <stdio.h>
#include <math.h>
#include <rasterfile.h>
#include "../include/struct.h"
#include "../include/constants.h"

/*****************************************************************************
 corner4() - Corner detector using DET

 Input - aImage         : Actual image
         LeftTopX       : x coordinate of left top corner of bounding box
         LeftTopY       : y coordinate of left top corner of bounding box
         RightBottomX   : x coordinate of right bottom corner of bounding box
         RightBottomY   : y coordinate of right bottom corner of bounding box
         PointNum       : Number of Feature points required
         Feature_Point  : Feature point data array
         image_width    : width of the input image

Process - It crop out the region defined by LeftTopX, LeftTopY, RightBottomX
          and RightBottomY first and performs Prewitt convolution on that
          region.  The result of Prewitt convolution is stored in mag_x and
          mag_y representing the gradient in x and y direction.  Then we apply
          Prewitt operator on mag_x and mag_y to obtain mag_xx, mag_xy and
          mag_yy.  Using these information, we can find DET at each pixel.

Output - None.

Return value - When fail to allocate memory, it returns -1.
               If it successfully processes the image, it returns 1.

Created by - Collins Chien on 2nd June, 1993.
             Modified to corner detection on 22nd November, 1993.

*****************************************************************************/
corner4(aImage, LeftTopX, LeftTopY, RightBottomX, RightBottomY, PointNum, Featur
e_Point, image_width)
int     *aImage;                /* The image */
int     LeftTopX, LeftTopY;     /* Left top corner of the bounding box */
int     RightBottomX, RightBottomY;     /* Right bottom corner of the bounding b
ox */
int     PointNum;               /* Number of Feature points required */
POINT   *Feature_Point;         /* Feature Point Data array */
int     image_width;
{
        int     *temp_mem, *fin_mem;
        int     *pixel_ptr;
        int     i, j, k, index;
        int     m, mm, index1, bound1, bound2, bound3, *pos, total;
        int     box_width, box_height;
        float   *mag_x, *mag_y, *mag_xx, *mag_yy, *mag_xy, *dd, center, f_min, s
_min;
        int     dd_temp,l,n,o;
        float   max_mag, min_mag;
        float   dx, dy, ddx, ddy, dd_max, p2;
        int     hist[256], Threshold, fcmp();
        POINT   pt[MAX_FEATURE_PT];
        int     calc_mag();
        float   dump, sigma, mean;
        int     iteration;

        iteration = 0;
        dd_max = 3.1415927/4.0;
        p2 = 3.1415927*2.0;

fprintf(stderr,"Number of feature point specified = %d\n", PointNum);

        box_width = RightBottomX - LeftTopX + 1;
        box_height = RightBottomY - LeftTopY + 1;
        bound1 = 2*box_width;
        bound2 = 3*box_width;
        bound3 = 4*box_width;
        total = box_width*box_height;

        if(box_width <= 0 || box_height <= 0) {
           fprintf(stderr, "Invalid bounding box!\n");
           return(-1);
        }
        temp_mem = (int *)calloc(box_width*box_height, sizeof(int));
        if(temp_mem == NULL) {
           fprintf(stderr, "Unable to create memory for the bounded image!\n");
           return(-1);
        }

/*************************************************************
 Copy bounding box image into temporary memory for processing
*************************************************************/
        pixel_ptr = aImage;
        for(i=0; i<box_height; i++) {
           for(j=0; j<box_width; j++) {
                index = image_width*(i+LeftTopY) + j+LeftTopX;
                temp_mem[i*box_width+j] = pixel_ptr[index];
           }
        }

        mag_x = (float *)calloc(box_width*box_height, sizeof(float));
        mag_y = (float *)calloc(box_width*box_height, sizeof(float));
        mag_xx = (float *)calloc(box_width*box_height, sizeof(float));
        mag_yy = (float *)calloc(box_width*box_height, sizeof(float));
        mag_xy = (float *)calloc(box_width*box_height, sizeof(float));

        if((mag_x==NULL)||(mag_y==NULL)||(mag_xx==NULL)||(mag_xy==NULL)||(mag_yy
==NULL)) {
           fprintf(stderr, "Unable to create memory for the magnitude map!\n");
           free(temp_mem);
           return(-1);
        }
        dd = (float *)calloc(box_width*box_height, sizeof(float));
        if(dd == NULL) {
           fprintf(stderr, "Unable to create memory for the double derivative ma
p!\n");
           free(mag_x);
           free(mag_y);
           free(mag_xx);
           free(mag_xy);
           free(mag_yy);
           free(temp_mem);
```

```
            return(-1);
        }

        pos = (int *)calloc(MAX_FEATURE_PT, sizeof(int));
        if(pos == NULL) {
            fprintf(stderr, "Unable to create memory for the position map!\n");
            free(mag_x);
            free(mag_y);
            free(mag_xx);
            free(mag_xy);
            free(mag_yy);
            free(temp_mem);
            free(dd);
            return(-1);
        }

/************************
 Initialize all the maps
************************/
        for(i=0; i<box_height; i++) {
            for(j=0; j<box_width; j++) {
                index = i*box_width+j;
                mag_x[index] = 0.0;
                mag_y[index] = 0.0;
                mag_xx[index] = 0.0;
                mag_xy[index] = 0.0;
                mag_yy[index] = 0.0;
                dd[index] = 0.0;
            }
        }
        for(i=0; i<256; i++)
            hist[i] = 0;
        for(i=0; i<MAX_FEATURE_PT; i++) {
            pos[i] = 0;
            pt[i].mag = 0;
            pt[i].x = 0;
            pt[i].y = 0;
            pt[i].index = 0;
        }

        max_mag = -100000.0;
        min_mag = 100000.0;

        for(i=1; i<box_height-1; i++)
            for(j=1; j<box_width-1; j++) {
                index = i*box_width + j;

/*****************************************************
 Perform Prewitt convolution on the bounded image first
*****************************************************/
                mag_y[index] = (float)(temp_mem[index-box_width-1]+temp_mem[inde
x-box_width]+temp_mem[index-box_width+1]-temp_mem[index+box_width-1]-temp_mem[in
dex+box_width]-temp_mem[index+box_width+1])/3.0;

                mag_x[index] = (float)(temp_mem[index-box_width-1]-temp_mem[inde
x-box_width+1]+temp_mem[index-1]-temp_mem[index+1]+temp_mem[index+box_width-1]-t
emp_mem[index+box_width+1])/3.0;
            } /* for loop */

        for(i=2; i<box_height-2; i++)
            for(j=2; j<box_width-2; j++) {
                index = i*box_width + j;
                mag_yy[index] = (float)(mag_y[index-box_width-1]+mag_y[index-box
_width]+mag_y[index-box_width+1]-mag_y[index+box_width-1]-mag_y[index+box_width]
-mag_y[index+box_width+1])/3.0;

                mag_xx[index] = (float)(mag_x[index-box_width-1]-mag_x[index-box
_width+1]+mag_x[index-1]-mag_x[index+1]+mag_x[index+box_width-1]-mag_x[index+box
_width+1])/3.0;

                mag_xy[index] = (float)(mag_x[index-box_width-1]+mag_x[index-box
_width]+mag_x[index-box_width+1]-mag_x[index+box_width-1]-mag_x[index+box_width]
-mag_x[index+box_width+1])/3.0;
            } /* for loop */

/***************
 Beaudet's DET
***************/
        for(i=2; i<box_height-2; i++)
            for(j=2; j<box_width-2; j++) {
                index = i*box_width + j;
                dd[index] = fabs(mag_xx[index]*mag_yy[index]-mag_xy[index]*mag_x
y[index]);

/*******************************
 Find the max and min magnitude
*******************************/
                if(dd[index] > max_mag)
                    max_mag = dd[index];
                if(dd[index] < min_mag)
                    min_mag = dd[index];

            }  /* for loop */

/*******************************************************
 Normalize the magnitude map to the range from 0 to 255
*******************************************************/
        free(mag_x);
        free(mag_y);
        free(mag_xx);
        free(mag_xy);
        free(mag_yy);

        fin_mem = (int *)calloc(box_width*box_height, sizeof(int));
        for(i=0; i<box_height; i++)
            for(j=0; j<box_width; j++) {
                index = i*box_width + j;
                fin_mem[index] = temp_mem[index];
            }

/*******************************************************
 Normalize the magnitude map to the range from 0 to 255
*******************************************************/
        for(i=2; i<box_height-2; i++)
            for(j=2; j<box_width-2; j++) {
                index = i*box_width + j;
```

```
                dd[index] = (dd[index]-min_mag)*255.0/(max_mag-min_mag);
                hist[(int)(dd[index])]++;
             }
/*      dump = 0.0;  m = 0;
        for(i=2; i<box_height-2; i++)
           for(j=2; j<box_width-2; j++) {
                index = i*box_width + j;
                dump += dd[index];
                m++;
           }
        mean = dump/(float)m;
        mean = mean*mean;
        dump = 0.0;
        for(i=2; i<box_height-2; i++)
           for(j=2; j<box_width-2; j++) {
                index = i*box_width + j;
                dump += dd[index]*dd[index] - mean;
           }
        sigma = sqrt(dump/(float)(m-1));
        Threshold = 255 - nint(sigma*3);
fprintf(stderr, "sigma = %.3lf\n", sigma);
*/

/*********************
 Find threshold value
*********************/
        m = 0;
/*      for(i=255; (i>=0) && (m<MAX_FEATURE_PT); i--)
           m += hist[i];
        if(m >= MAX_FEATURE_PT)
           Threshold = i;
        else
           Threshold = i+1;
*/
incr:   m = 0; Threshold = 255;
   for(m=0; (m<PointNum) && (Threshold>0); ) {
        for(i=2; i<box_height-2; i++)
           for(j=2; j<box_width-2; j++) {
                index = i*box_width + j;
                if((nint(dd[index]) == Threshold) && (m < MAX_FEATURE_PT)) {
                   pos[m] = index;
                   pt[m].x = LeftTopX+j;
                   pt[m].y = LeftTopY+i;
                   pt[m].mag = dd[index];
                   pt[m].index = index;
                   m++;
                }
           }
           Threshold--;
   }


        mm = m;
/*******************************
 Suppression of local non-maxima
*******************************/
        for(i=0; i<m; i++) {
           index = pos[i];
           if( ((index - bound1) > 0) && ((index + bound1) < total) ) {
/***********
 3x3 window
************/
                for(j=-1; j<2 && (dd[index] > 0); j++)
                   for(k=-1; k<2 && (dd[index] > 0); k++) {
                        index1 = index + j*box_width + k;
                        if( dd[index1] > dd[index] ) {
                           dd[index] = 0;
                           pt[i].mag = 0.0;
                           mm--;
                        }
                   }
           }
           if( ((index - bound2) > 0) && ((index + bound2) < total) && (dd[index
] > 0) ) {
/***********
 5x5 window
************/
                for(j=-2; j<3 && (dd[index] > 0); j++)
                   for(k=-2; k<3 && (dd[index] > 0); k++) {
                        index1 = index + j*box_width + k;
                        if( dd[index1] > dd[index] ) {
                           dd[index] = 0;
                           pt[i].mag = 0.0;
                           mm--;
                        }
                   }
           }
           if( ((index - bound3) > 0) && ((index + bound3) < total) && (dd[index
] > 0) ) {
/***********
 7x7 window
************/
                for(j=-3; j<4 && (dd[index] > 0); j++)
                   for(k=-3; k<4 && (dd[index] > 0); k++) {
                        index1 = index + j*box_width + k;
                        if( dd[index1] > dd[index] ) {
                           dd[index] = 0;
                           pt[i].mag = 0.0;
                           mm--;
                        }
                   }
           }

        }
fprintf(stderr, "# of interest points after 2nd operation = %d\n", mm);
        iteration++;
        if(iteration < 4) goto incr;


/*********************************************
 Sort the Feature_Point array in decending order
*********************************************/
        qsort(pt, MAX_FEATURE_PT, sizeof(pt[0]), fcmp);

        for(i=0; i<PointNum; i++) {
           Feature_Point[i].x = pt[i].x;
```

```
            Feature_Point[i].y = pt[i].y;
            Feature_Point[i].mag = pt[i].mag;
            Feature_Point[i].index = pt[i].index;
        }

fprintf(stdout, "Threshold = %d\n", Threshold);

/************************
 Put crosses in the image
*************************/
        for(i=0; i<PointNum; i++) {
            if( pt[i].mag > 0 )
                draw_cross(fin_mem, temp_mem, pt[i].index, box_width, 5);
        }


/*********************************************************
 Find zero crossings and threshold according to magnitude
*********************************************************/
        for(i=1; i<box_height-1; i++)
            for(j=1; j<box_width-1; j++) {
                index = i*box_width + j;
                pixel_ptr[image_width*(i+LeftTopY)+LeftTopX+j] = fin_mem[index];
            }  /* for loop */

        free(temp_mem);
        free(fin_mem);
        free(dd);
        free(pos);
        free(mag_x);
        free(mag_y);
        free(mag_xx);
        free(mag_xy);
        free(mag_yy);
        return(1);
}
```

```
#include <stdio.h>
#include <math.h>
#include <rasterfile.h>
#include "../include/struct.h"
#include "../include/constants.h"

extern POINT *f_pt1, *f_ptw;
extern int f_pt_num1, f_pt_numw;
extern IMAGE im1, imw;
extern POINT from1, fromw, to1, tow;
int mask_side;


/*********************************************************************
 match3() - Multiresolution image matching algorithm

 Input - Image1         First image
         width1         Image width of first image
         height1        Image height of first image
         Image2         Second image
         width2         Image width of second image
         height2        Image height of second image
         Dx             Horizontal translation movement
         Dy             Vertical translation movement
         Theta          Rotational movement
         Scale          scaling factor

 Procedure -    1. Take the original images and reduce them to the lowest
                   resolution
                2. Apply feature detection in both images
                3. Apply initial transformation parameter on the lowest
                   resolution version of frame t1 and its feature points
                3. Do point matching by normalized correlation
                4. Find Dx', Dy', Theta' and Scale' (adjustment)
                5. Update Dx, Dy, Theta, Scale
                6. Reduce the image resolution corresponding to current
                   layer of the matching hierarchy
                7. Magnify the coordinates of the feature points
                   corresponding to the resolution of the current layer
                8. Apply an affine transform with parameter on frame t1 and
                   its feature points
                9. Do matching refinement to obtain Dx', Dy', Theta', Scale'
               10. Upate the estimates
               11. If current level is at the highest resolution then stop
                      otherwise, increase image resolution and adjust the
                      translation estimation by
                        2Dx => Dx
                        2Dy => Dy
                      goto 6.

 Output - None

 Procedure calls -      REDUCE()
                        corner10()
                        AFFINE_TRANSFORM()
                        AT_FP()
                        MERGE_LIST()
                        UPDATE_PARAMETERS()

 Return value - When fail to allocate memory, it returns -1.
                If it successfully processes the image, it returns 1.

 Created by - Collins Chien on 8th June, 1994.

*********************************************************************/
match3(Image1, width1, height1, Image2, width2, height2, Dx, Dy, Theta, Scale)
int *Image1, *Image2;
int width1, width2, height1, height2;
int *Dx, *Dy;
float *Theta, *Scale;
{
        int i, j, index;
        int *cur1, *cur2;
        int iwidth1, iwidth2, iheight1, iheight2;
        int lowest_level1;
        int lowest_level2, level;
        int *REDUCE(), *AFFINE_TRANSFORM();
        extern int corner10();
        extern POINT *match1_pt, *matchw_pt;
        int AT_FP();
        int num, num1, numw;
        POINT *tmp_pt, *matched_pt, *MERGE_LIST();
        int INITIAL_MATCH(), MATCH_REFINE();
        int matched_num;
        float tmp_scale;
        float tmp_theta;
        int tmp_dx, tmp_dy;
        float fdx, fdy;
        int MAGNIFY_FP();
        extern void overlap();
        POINT f1, t1, fw, tw;
        int b, w;
        char fname1[20], fname2[20];
        extern int write_im();
        int lim;
        POINT *VERIFY();
        float tmp_x, tmp_y, dist;
        extern int match();

        mask_side = 8;
        lim = 200;
/*      overlap(); */
        tmp_scale = *Scale;
        tmp_theta = *Theta;
        tmp_dx = -(*Dx);
        tmp_dy = -(*Dy);
        iwidth1 = width1; iheight1 = height1;
        iwidth2 = width2; iheight2 = height2;
        for(i=0; iwidth1>lim; i++)
           iwidth1 = iwidth1/2;
        for(j=0; iheight1>lim; j++)
           iheight1 = iheight1/2;
        lowest_level1 = MIN(i,j);

        for(i=0; iwidth2>lim; i++)
           iwidth2 = iwidth2/2;
```

```
        for(j=0; iheight2>lim; j++)
           iheight2 = iheight2/2;
        lowest_level2 = MIN(i,j);

        level = MIN(lowest_level1, lowest_level2);

        f1.x = from1.x; f1.y = from1.y;
        t1.x = to1.x;   t1.y = to1.y;
        fw.x = fromw.x; fw.y = fromw.y;
        tw.x = tow.x;   tw.y = tow.y;
        fdx = (float)tmp_dx;
        fdy = (float)tmp_dy;
        for(i=0; i<level; i++) {
           fdx /= 2.0;
           fdy /= 2.0;
           f1.x /= 2; f1.y /= 2;
           t1.x /= 2; t1.y /= 2;
           fw.x /= 2; fw.y /= 2;
           tw.x /= 2; tw.y /= 2;
           mask_side *= 2;
        }
        tmp_dx = nint(fdx);
        tmp_dy = nint(fdy);

        fprintf(stdout, "Level = %d\n", level);
        fprintf(stdout, "Image #1 initial width = %d  height = %d\n", iwidth1, i
height1);
        fprintf(stdout, "Image #2 initial width = %d  height = %d\n", iwidth2, i
height2);
/*************************************
 Reduce both images to lowest resolution
*************************************/
        fprintf(stdout, "Reducing images.");
        cur1 = REDUCE(Image1, width1, height1, level, &iwidth1, &iheight1);
        fprintf(stdout, ".");
        cur2 = REDUCE(Image2, width2, height2, level, &iwidth2, &iheight2);
        fprintf(stdout, ".");
        if((cur1 == NULL) || (cur2 == NULL)) {
           fprintf(stderr, "Cannot create memory for cur1 or cur2 in match3()\n"
);
           return(-1);
        }
        sprintf(fname1, "t1_%d.ras\0", level);
        sprintf(fname2, "t2_%d.ras\0", level);
        write_im(fname1, cur1, iwidth1, iheight1, &w, iheight1, 0, 0, INT_IM);
        write_im(fname2, cur2, iwidth2, iheight2, &w, iheight2, 0, 0, INT_IM);


/*************************************
 Find feature points out of both images
*************************************/
        b = nint(.05*(float)MIN(iwidth1, iheight1));
        fprintf(stdout, ".Finding Feature Points.");
        free(f_pt1);
        f_pt1 = (POINT *)calloc(f_pt_num1, sizeof(POINT));
        corner10(cur1, f1.x+b, f1.y+b, t1.x-1-b, t1.y-1-b, f_pt_num1, f_pt1, iwi
dth1);
        b = nint(.05*(float)MIN(iwidth2, iheight2));
        free(f_ptw);
        fprintf(stdout, ".");
        f_ptw = (POINT *)calloc(f_pt_numw, sizeof(POINT));
        corner10(cur2, fw.x+b, fw.y+b, tw.x-1-b, tw.y-1-b, f_pt_numw, f_ptw, iwi
dth2);
        fprintf(stdout, ".");

/*************************************
 Match feature points of two images
*************************************/
        free(match1_pt);
        free(matchw_pt);
        tmp_pt = f_pt1;
        for(num1=0; (num1 < f_pt_num1) && (tmp_pt->mag > 0); num1++)
           tmp_pt++;
        tmp_pt = f_ptw;
        for(numw=0; (numw < f_pt_numw) && (tmp_pt->mag > 0); numw++)
           tmp_pt++;

/*****************************************************
 Do affine transform on image 2 and its feature points
*****************************************************/
        fprintf(stdout, "\n Affine transform.");
        AT_FP(f_ptw, numw, iwidth2, iheight2, tmp_theta, tmp_scale);
        fprintf(stdout, ".");
        cur2 = AFFINE_TRANSFORM(cur2, &iwidth2, &iheight2, tmp_theta, tmp_scale)
;
        fprintf(stdout, ".");
        if(cur2 == NULL)
           return(-1);

/*      num = (num1 > numw ? num1 : numw);
        match1_pt = (POINT *)calloc(num, sizeof(POINT));
        matchw_pt = (POINT *)calloc(num, sizeof(POINT));
*/
        match1_pt = (POINT *)calloc(num1, sizeof(POINT));
        matchw_pt = (POINT *)calloc(numw, sizeof(POINT));

        if((match1_pt==NULL) || (matchw_pt==NULL)) {
           fprintf(stderr, "Cannot allocate memory for match1_pt or matchw_pt\n"
);
           free(match1_pt);
           free(matchw_pt);
           exit(-1);
        }
        for(i=0; i<num1; i++)
           fprintf(stdout, "%d. (%d, %d)\n", i, f_pt1[i].x, f_pt1[i].y);
        for(i=0; i<numw; i++)
           fprintf(stdout, "%d. (%d, %d)\n", i, f_ptw[i].x, f_ptw[i].y);
        fprintf(stdout, ".Matching features.\n");
        INITIAL_MATCH(cur1, iwidth1, iheight1, f_pt1, num1, match1_pt, cur2, iwi
dth2, iheight2, f_ptw, numw);
        fprintf(stdout, ".");

/*      INITIAL_MATCH(cur2, iwidth2, iheight2, f_ptw, numw, matchw_pt, cur1, iwi
dth1, iheight1, f_pt1, num1);
        fprintf(stdout, ".\n");
        matched_pt = MERGE_LIST(f_pt1, match1_pt, num1, f_ptw, matchw_pt, numw,
```

```
&matched_num); */

/*      match(f_pt1, num1, f_ptw, numw, match1_pt, matchw_pt, &num, im1, imw);
*/      fprintf(stdout, ".\n");

        matched_num = num1*2;

/*      matched_num = num*2;
*/      matched_pt = (POINT *)calloc(matched_num, sizeof(POINT));
        if(matched_pt == NULL) {
           fprintf(stderr, "Cannot allocate memory for matched_pt in match3()\n"
);
           return(-1);
        }
        for(i=0; i<num1; i++) {
/*      for(i=0; i<num; i++) {
*/         index = 2*i;
           matched_pt[index].x = f_pt1[i].x;
           matched_pt[index].y = f_pt1[i].y;
           matched_pt[index].mag = f_pt1[i].mag;

           matched_pt[index+1].x = match1_pt[i].x;
           matched_pt[index+1].y = match1_pt[i].y;
           matched_pt[index+1].mag = match1_pt[i].mag;
/*         matched_pt[index+1].x = matchw_pt[i].x;
           matched_pt[index+1].y = matchw_pt[i].y;
           matched_pt[index+1].mag = matchw_pt[i].mag;
*/      }
        for(i=0; i<matched_num/2; i++)
           fprintf(stdout, "(%d, %d) -> (%d, %d) mag: %.3lf\n", matched_pt[2*i].
x, matched_pt[2*i].y, matched_pt[2*i+1].x, matched_pt[2*i+1].y, matched_pt[2*i].
mag);

        fprintf(stdout, "Finish matching...\n");

        matched_pt = VERIFY(matched_pt, &matched_num, level);
        if(matched_pt == NULL)
           return(-1);

        for(i=0; i<matched_num/2; i++)
           fprintf(stdout, "(%d, %d) -> (%d, %d) mag: %.3lf\n", matched_pt[2*i].
x, matched_pt[2*i].y, matched_pt[2*i+1].x, matched_pt[2*i+1].y, matched_pt[2*i].
mag);
        free(match1_pt);
        free(matchw_pt);
        free(f_pt1);
        free(f_ptw);

/*********************************
 Update transformation parameters
*********************************/
        fprintf(stdout, ".Update parameters.\n");
        if(UPDATE_PARAMETERS(matched_pt, matched_num, &tmp_theta, &tmp_scale, &t
mp_dx, &tmp_dy) == -1)
           return(-1);
        fprintf(stdout, ".\n");

        fprintf(stdout, "Theta = %.3lf, Scale = %.3lf\n", tmp_theta, tmp_scale);
        fprintf(stdout, "Dx = %d, Dy = %d\n", tmp_dx, tmp_dy);
        fprintf(stdout, "Finish initial matching...\n\n");

   while(level > 0) {
/*************************************************************************
 Magnify feature points coordinate corresponding to current resolution
*************************************************************************/
        tmp_dx *= 2.0;
        tmp_dy *= 2.0;
        fprintf(stdout, "Magnifying feature points.");
        MAGNIFY_FP(matched_pt, matched_num);
        fprintf(stdout, ".");
        if(fabs(tmp_theta*RADIAN2DEGREE) < 1.0) tmp_theta = 0;
/*      if(fabs(tmp_scale-1) < 0.1) tmp_scale = 1;  */

/*******************************
 Go one level up in the pyramid
*******************************/
        level--;
        free(cur1);
        free(cur2);
        fprintf(stdout, "Reducing images.");
        cur1 = REDUCE(Image1, width1, height1, level, &iwidth1, &iheight1);
        fprintf(stdout, ".\n");
        cur2 = REDUCE(Image2, width2, height2, level, &iwidth2, &iheight2);
        if((cur1 == NULL) || (cur2 == NULL)) {
           fprintf(stderr, "Cannot create memory for cur1 or cur2 in match3()\n"
);
           return(-1);
        }
        if(level > 0) {
        sprintf(fname1, "t1_%d.ras\0", level);
        sprintf(fname2, "t2_%d.ras\0", level);
        write_im(fname1, cur1, iwidth1, iheight1, &w, iheight1, 0, 0, INT_IM);
        write_im(fname2, cur2, iwidth2, iheight2, &w, iheight2, 0, 0, INT_IM);
        }
        fprintf(stdout, "Level = %d\n", level);
        fprintf(stdout, "Image #1 width = %d  height = %d\n", iwidth1, iheight1)
;
        fprintf(stdout, "Image #2 width = %d  height = %d\n", iwidth2, iheight2)
;

/*******************************
 Do affine transform on image 2
*******************************/
        fprintf(stdout, "Affine transform.");
        cur2 = AFFINE_TRANSFORM(cur2, &iwidth2, &iheight2, tmp_theta, tmp_scale)
;
        fprintf(stdout, ".");
        if(cur2 == NULL)
           return(-1);
/*****************
 Match refinement
*****************/
        fprintf(stdout, "Match refine.");
        MATCH_REFINE(cur1, iwidth1, iheight1, cur2, iwidth2, iheight2, matched_p
t, matched_num, -tmp_dx, -tmp_dy, level);
```

```
/*      matched_pt = VERIFY(matched_pt, &matched_num, level); */

        for(i=0; i<matched_num/2; i++) {
           tmp_x = (float)abs(matched_pt[2*i].x-matched_pt[2*i+1].x);
           tmp_y = (float)abs(matched_pt[2*i].y-matched_pt[2*i+1].y);
           dist = hypot(tmp_x, tmp_y);
           fprintf(stdout, "(%d, %d) -> (%d, %d) mag: %.3lf  err: %.3lf\n", matc
hed_pt[2*i].x, matched_pt[2*i].y, matched_pt[2*i+1].x, matched_pt[2*i+1].y, matc
hed_pt[2*i].mag, dist);
        }

/************ ******************
 Update transformation parameters
**********************************/
        fprintf(stdout, "Upate parameters.");
        if(UPDATE_PARAMETERS(matched_pt, matched_num, &tmp_theta, &tmp_scale, &t
mp_dx, &tmp_dy) == -1)
           return(-1);
        fprintf(stdout, ".\n");
        fprintf(stdout, "Level = %d\n", level);
        fprintf(stdout, "Theta = %.3lf, Scale = %.3lf\n", tmp_theta, tmp_scale);
        fprintf(stdout, "Dx = %d, Dy = %d\n\n", tmp_dx, tmp_dy);

   }
        *Theta = tmp_theta;
        *Scale = tmp_scale;
        *Dx = -tmp_dx;
        *Dy = -tmp_dy;
        fprintf(stdout, "Exiting match3()\n");
        free(cur1);
        free(cur2);
        free(matched_pt);
        free(match1_pt);
        free(f_pt1);
        free(tmp_pt);

        return(1);
}


/*********************************************************************
 REDUCE() - reduce image to lower resolution

 Input - image          original image
         width          width of the original image
         height         height of the original image
         level          level of image in the pyramid
         new_width      reduced width
         new_height     reduced height

 Output - None

 Procedure calls -      warp_image()

 Return - Integer pointer of the reduced image
          new width and height of the reduced image

 Created by - Collins Chien on June 9th, 1994
*********************************************************************/
int *REDUCE(image, width, height, level, new_width, new_height)
int *image, width, height;
int level;
int *new_width, *new_height;
{
        int wodd, hodd, iheight, iwidth, *cur;
        int i, j, k, l, index, index1;
        float fwidth, fheight;
        extern int warp_image();
        POINT warp_pt[4];
        int step, sum, num, x, y;


        fwidth = (float)width;
        fheight = (float)height;
        step = 1;

        if((image == NULL) || (level < 0))
           return(NULL);

        for(i=0; i< level; i++) {
           fwidth = fwidth/2.0;
           fheight = fheight/2.0;
           step *= 2;
        }

        iwidth = nint(fwidth);
        iheight = nint(fheight);
        *new_width = iwidth;
        *new_height = iheight;

        cur = (int *)calloc((iwidth*iheight), sizeof(int));

        for(i=0; i<iheight; i++)
           for(j=0; j<iwidth; j++) {
                index = i*iwidth+j;
                sum = 0;
                num = 0;
                for(k=0; k<step; k++)
                   for(l=0; l<step; l++) {
                        x = j*step+l;
                        y = i*step+k;
                        if((x < width) && (y < height)) {
                           index1= y*width+x;
                           sum += image[index1];
                           num++;
                        }
                   }
                cur[index] = sum/num;
           }

/*      warp_pt[0].x = iwidth-1;        warp_pt[0].y = 0;
        warp_pt[1].x = 0;       warp_pt[1].y = 0;
        warp_pt[2].x = 0;       warp_pt[2].y = iheight-1;
        warp_pt[3].x = iwidth-1;        warp_pt[3].y = iheight-1;
```

```
        warp_image(image, width, height, cur, iwidth, iheight, warp_pt, -1); */

        return(cur);
}

/*****************************************************************
 AFFINE_TRANSFORM() - do affine transformation on an image

 Input - image          original image
         width          width of the original image
         height         height of the original image
         theta          rotational angle
         scale          scaling factor

 Output - None

 Procedure calls -      warp_image()

 Return - Integer pointer of the transformed image
          width and height will be changed to the new ones.

 Note: This routine only does rotation and scaling

 Created by - Collins Chien on June 10th, 1994

*****************************************************************/
int *AFFINE_TRANSFORM(image, width, height, theta, scale)
int *image;
int *width, *height;
float theta, scale;
{
        int i, j;
        POINT warp_pt[4];
        float tmp[4][2], n_tmp[4][2], tmp_w, tmp_h;
        float s_t, c_t, t;
        float minx, miny, maxx, maxy;
        int *new_image, new_width, new_height, size;
        extern warp_image();

        if((image == NULL) || ((theta == 0) && (scale == 1))) {
           size = (*width)*(*height);
           new_image = (int *)calloc(size, sizeof(int));
           if(new_image == NULL) {
                fprintf(stderr, "Cannot create new_image in AFFINE_TRANSFORM()\n
");
                return(image);
           }
           for(i=0; i<size; i++)
                new_image[i] = image[i];
           free(image);
           return(new_image);
        }

        tmp_w = (float)(*width)/2.0;
        tmp_h = (float)(*height)/2.0;
        tmp[0][0] = tmp_w; tmp[0][1] = -tmp_h;
        tmp[1][0] = -tmp_w; tmp[1][1] = -tmp_h;
        tmp[2][0] = -tmp_w; tmp[2][1] = tmp_h;
        tmp[3][0] = tmp_w; tmp[3][1] = tmp_h;
        t = theta;
        s_t = sin(t);
        c_t = cos(t);

/*************************************************
 Calculate the affine transform of the four corners
*************************************************/
        for(i=0; i<4; i++) {
           n_tmp[i][0] = scale*(tmp[i][0]*c_t + tmp[i][1]*s_t);
           n_tmp[i][1] = scale*(-tmp[i][0]*s_t + tmp[i][1]*c_t);
        }

        for(i=0;i<4;i++)
           printf("n_tmp[%d][0] = %f, n_tmp[%d][1] = %f\n",i,n_tmp[i][0],i,n_tmp
[i][1]);

        minx = n_tmp[0][0]; miny = n_tmp[0][1];
        maxx = n_tmp[0][0]; maxy = n_tmp[0][1];

        /*********************************************************************
         * calculate the dimension of the bounding box to contain the image *
         *********************************************************************/
        for(i=1; i<4; i++) {
           if(maxx < n_tmp[i][0])
                maxx = n_tmp[i][0];
           if(minx > n_tmp[i][0])
                minx = n_tmp[i][0];
           if(maxy < n_tmp[i][1])
                maxy = n_tmp[i][1];
           if(miny > n_tmp[i][1])
                miny = n_tmp[i][1];
        }
        printf("maxx = %f, maxy = %f, minx = %f, miny = %f\n",maxx,maxy,minx,min
y);

        /*************************************************************
         * assign the four corners into the array for warping later *
         *************************************************************/
        for(i=0; i<4; i++) {
           warp_pt[i].x = nint(n_tmp[i][0] - minx);
           warp_pt[i].y = nint(n_tmp[i][1] - miny);
           printf("warp_pt[%d].x = %d, warp_pt[%d].y = %d\n",i,warp_pt[i].x,i,wa
rp_pt[i].y);
        }
        new_width = nint(maxx-minx+1);
        new_height = nint(maxy-miny+1);
        new_image = (int *)calloc(new_width*new_height, sizeof(int));
        if(new_image == NULL) {
           fprintf(stderr, "Cannot create new_image in AFFINE_TRANSFORM()\n");
           return(image);
        }
        warp_image(image, *width, *height, new_image, new_width, new_height, war
p_pt, -1);
        *width = new_width;
        *height = new_height;
        free(image);
        return(new_image);
```

```
}

/*********************************************************************
 AT_FP() - Affine Transform Feature Points

 Input - f_pt           feature points array
         theta          rotational angle
         scale          scaling factor

 Output - None

 Procedure calls - None

 Return - 1 if success, otherwise -1

 Created by - Collins Chien on June 10th, 1994

*********************************************************************/
int AT_FP(f_pt, num, width, height, theta, scale)
POINT *f_pt;
int width, height, num;
float theta, scale;
{
        int i;
        float c_t, s_t, t;
        float tmp_x, tmp_y, tmp_w, tmp_h;
        POINT *ff_pt;

        ff_pt = f_pt;
        if((ff_pt == NULL) || ((theta == 0) && (scale == 1)))
           return(1);
        t = theta;
        c_t = cos(t);
        s_t = sin(t);
        tmp_w = (float)width/2;
        tmp_h = (float)height/2;

        while((i < num) || (ff_pt->mag != 0)) {
           tmp_x = (float)(ff_pt->x) - tmp_w;
           tmp_y = (float)(ff_pt->y) - tmp_h;
           ff_pt->x = nint(scale*(tmp_x*c_t + tmp_y*s_t) + tmp_w);
           ff_pt->y = nint(scale*(-tmp_x*s_t + tmp_y*c_t) + tmp_h);
           ff_pt++;
           i++;
        }
        return(1);
}

/*********************************************************************
 MATCH_REFINE() - Refining Matched Feature Points

 Input - image1         first image
         width1         width of first image
         height1        height of first image
         image2         second image
         width2         width of second image
         height2        height of second image
         matched_pt     matched feature points array
         matched_num    number of matched points
         dx             horizontal translation
         dy             vertical translation

 Output - None

 Procedure calls - FIND_SAVD()

 Return - 1 if success, otherwise -1

 Created by - Collins Chien on June 16th, 1994

*********************************************************************/
int MATCH_REFINE(image1, width1, height1, image2, width2, height2, matched_pt, m
atched_num, dx, dy, level)
int *image1, *image2;
int width1, width2;
int height1, height2;
POINT *matched_pt;
int matched_num;
int dx, dy;
int level;
{
        int i, j, k, index;
        int WS;
        int x, y;
        float val, min_val;
        float FIND_COR();
        float FIND_COM();
        float FIND_SAVD();
        float FIND_SSC();
        int temp_num;
        float sum, num;

/***************
 Initialization
***************/
        WS = 1;
        temp_num = matched_num/2;
        k = 0;

        while(k < temp_num) {
           sum = 0.0;
           num = 0.0;
           min_val = 100000;
           for(i=-WS; i<= WS; i++)
                for(j=-WS; j<=WS; j++) {
                   index = 2*k;
                   x = (matched_pt[index].x)+j;
                   y = (matched_pt[index].y)+i;
                   if((x < 0) || (x > width1) || (y < 0) || (y > height1))
                        continue;
                   val = FIND_SAVD(image1, width1, height1, x, y, image2, width2
, height2, dx, dy, level);
/*                 sum += val;
                   num++; */
                   if(val < min_val) {
                        min_val = val;
```

```
                    index = 2*k+1;
                    matched_pt[index].x = x;
                    matched_pt[index].y = y;
                    matched_pt[index].mag = min_val;
                    matched_pt[index-1].mag = min_val;
                }
            }
/*          sum /= num;
            min_val = 100000;
            for(i=-WS; i<= WS; i++)
                for(j=-WS; j<=WS; j++) {
                    index = 2*k;
                    x = (matched_pt[index].x)+j;
                    y = (matched_pt[index].y)+i;
                    if((x < 0) || (x > width1) || (y < 0) || (y > height1))
                        continue;
                    val = FIND_COM(image1, width1, height1, x, y, image2, width2,
 height2, dx, dy, level);
                    if(fabs(val-sum) < min_val) {
                        min_val = fabs(val-sum);
                        index = 2*k+1;
                        matched_pt[index].x = x;
                        matched_pt[index].y = y;
                        matched_pt[index].mag = val;
                        matched_pt[index-1].mag = val;
                    }
                } */

            k++;
        }
}

/*****************************************************************
 FIND_SSC() - Find stochastic sign change

 Input - image1          first image
         width1          width of first image
         x               x coordinate of the feature point
         y               y coordinate of the feature point
         image2          second image
         width2          width of second image
         dx              horizontal translation
         dy              vertical translation

 Output - None

 Procedure calls - None

 Return - Number of zero crossing

 Created by - Collins Chien on June 21st, 1994

*****************************************************************/
float FIND_SSC(image1, width1, height1, x, y, image2, width2, height2, dx, dy, l
evel)
int *image1, *image2;
int width1, width2;
int height1, height2;
int x, y;
int dx, dy;
int level;
{
        int i, j, index1, index2;
        int val, pre_val;
        int WM, num;

        WM = 8;
        pre_val = 0;
        num = 0;
/*      WM = mask_side;
        for(i=0; i<level; i++)
           WM /= 2; */
        for(i=-WM; i<=WM; i++)
           for(j=-WM; j<=WM; j++) {
                index1 = (i+y)*width1+j+x;
                index2 = (i+y-dy)*width2+j+x-dx;
                if( ((i+y)<0) || ((i+y)>height1) || ((j+x)<0) || ((j+x)>width1)
|| ((i+y-dy)<0) || ((i+y-dy)>height2) || ((j+x-dx)<0) || ((j+x-dx)>width2) || (i
mage1[index1] < 0) || (image2[index2] < 0) ) continue;
                val = image1[index1] - image2[index2];
                if( (val*pre_val) < 0)
                   num++;
                pre_val = val;
           }
        return((float)num);
}

/*****************************************************************
 FIND_SAVD() - Find sum of absolute valued differences SAVD

 Input - image1          first image
         width1          width of first image
         x               x coordinate of the feature point
         y               y coordinate of the feature point
         image2          second image
         width2          width of second image
         dx              horizontal translation
         dy              vertical translation

 Output - None

 Procedure calls - None

 Return - SAVD

 Created by - Collins Chien on June 21st, 1994

*****************************************************************/
float FIND_SAVD(image1, width1, height1, x, y, image2, width2, height2, dx, dy,
level)
int *image1, *image2;
int width1, width2;
int height1, height2;
int x, y;
int dx, dy;
int level;
```

```
{

        int i, j, index1, index2;
        int sum;
        int WM;

        sum = 0;
        WM = 8;
/*      WM = mask_side;
        for(i=0; i<level; i++)
           WM /= 2; */
        for(i=-WM; i<=WM; i++)
           for(j=-WM; j<=WM; j++) {
                index1 = (i+y)*width1+j+x;
                index2 = (i+y-dy)*width2+j+x-dx;
                if( ((i+y)<0) || ((i+y)>height1) || ((j+x)<0) || ((j+x)>width1)
|| ((i+y-dy)<0) || ((i+y-dy)>height2) || ((j+x-dx)<0) || ((j+x-dx)>width2) || (i
mage1[index1] < 0) || (image2[index2] < 0) ) continue;
                sum += abs(image1[index1] - image2[index2]);
           }
        return((float)sum);
}

/*****************************************************************
 FIND_COM() - Find combinational value

 Input - image1          first image
         width1          width of first image
         x               x coordinate of the feature point
         y               y coordinate of the feature point
         image2          second image
         width2          width of second image
         dx              horizontal translation
         dy              vertical translation

 Output - None

 Procedure calls - None

 Return - The combinational value

 Created by - Collins Chien on June 19th, 1994

*****************************************************************/
float FIND_COM(image1, width1, height1, x, y, image2, width2, height2, dx, dy, l
evel)
int *image1, *image2;
int width1, width2;
int height1, height2;
int x, y;
int dx, dy;
int level;
{

        int i, j, k, index1, index2;
        float mean1, mean2, mean1_sq, mean2_sq;
        float cov1, cov2;
        float size2;
        int num;
        float sim;
        int WM, total, in;
        POINT *com;

/*      WM = mask_side;
        for(i=0; i<level; i++)
           WM /= 2; */
        WM = 8;
        in = 0;
        total = (2*WM+1)*(2*WM+1);
        com = (POINT *)calloc(total, sizeof(POINT));
        for(i=0; i<total; i++) {
           com[i].x = -1;
           com[i].y = -1;
           com[i].mag = 0;
        }
        num = 0;
        for(i=-WM; i<=WM; i++)
           for(j=-WM; j<=WM; j++) {
                index1 = (i+y)*width1+j+x;
                index2 = (i+y-dy)*width2+j+x-dx;
                if( ((i+y)<0) || ((i+y)>height1) || ((j+x)<0) || ((j+x)>width1)
|| ((i+y-dy)<0) || ((i+y-dy)>height2) || ((j+x-dx)<0) || ((j+x-dx)>width2) || (i
mage1[index1] < 0) || (image2[index2] < 0) ) continue;
                for(k=0; (k<total) && (com[k].x != -1); k++) {
                   if((image1[index1] == com[k].x! && (image2[index2] == com[k].
y)) {
                        in = 1;
                        break;
                   }
                }
                if(in == 0) {
                   com[num].x = image1[index1];
                   com[num].y = image2[index2];
                   num++;
                }
                in = 0;
           }
        free(com);
        return((float)num);
}

/*****************************************************************
 FIND_COR() - Find correlation value

 Input - image1          first image
         width1          width of first image
         x               x coordinate of the feature point
         y               y coordinate of the feature point
         image2          second image
         width2          width of second image
         dx              horizontal translation
         dy              vertical translation

 Output - None

 Procedure calls - None
```

```
 Return - correlation value, -1 if fail to create memory

 Created by - Collins Chien on June 10th, 1994

*********************************************************************/
float FIND_COR(image1, width1, height1, x, y, image2, width2, height2, dx, dy, l
evel)
int *image1, *image2;
int width1, width2;
int height1, height2;
int x, y;
int dx, dy;
int level;
{

        int i, j, index1, index2;
        float mean1, mean2, mean1_sq, mean2_sq;
        float cov1, cov2;
        float size2;
        float num, sum1, sum2;
        float sim;
        int WM;

/*      WM = mask_side;
        for(i=0; i<level; i++)
           WM /= 2; */
        WM = 8;
        num = 0; sum1 = 0; sum2 = 0;
        for(i=-WM; i<=WM; i++)
                for(j=-WM; j<=WM; j++) {
                   index1 = (i+y)*width1+j+x;
                   index2 = (i+y-dy)*width2+j+x-dx;
                   if( ((i+y)<0) || ((i+y)>height1) || ((j+x)<0) || ((j+x)>width
1) || ((i+y-dy)<0) || ((i+y-dy)>height2) || ((j+x-dx)<0) || ((j+x-dx)>width2) ||
 (image1[index1] < 0) || (image2[index2] < 0) ) continue;
                   sum1 += (float)image1[index1];
                   sum2 += (float)image2[index2];
                   num++;
                }
           mean1 = sum1/num;
           mean2 = sum2/num;
           sum1 = 0; sum2 = 0;
           mean1_sq = mean1*mean1;
           mean2_sq = mean2*mean2;
           for(i=-WM; i<=WM; i++)
                for(j=-WM; j<=WM; j++) {
                   index1 = (i+y)*width1+j+x;
                   index2 = (i+y-dy)*width2+j+x-dx;
                   if( ((i+y)<0) || ((i+y)>height1) || ((j+x)<0) || ((j+x)>width
1) || ((i+y-dy)<0) || ((i+y-dy)>height2) || ((j+x-dx)<0) || ((j+x-dx)>width2) ||
 (image1[index1] < 0) || (image2[index2] < 0) ) continue;
                   sum1 += (float)(image1[index1]*image1[index1]) - mean1_sq;
                   sum2 += (float)(image2[index2]*image2[index2]) - mean2_sq;
                }
           cov1 = sqrt(fabs(sum1/(num-1)));
           cov2 = sqrt(fabs(sum2/(num-1)));
           sum1 = 0; sum2 = 0;
           for(i=-WM; i<=WM; i++)
                for(j=-WM; j<=WM; j++) {
                   index1 = (i+y)*width1+j+x;
                   index2 = (i+y-dy)*width2+j+x-dx;
                   if( ((i+y)<0) || ((i+y)>height1) || ((j+x)<0) || ((j+x)>width
1) || ((i+y-dy)<0) || ((i+y-dy)>height2) || ((j+x-dx)<0) || ((j+x-dx)>width2) ||
 (image1[index1] < 0) || (image2[index2] < 0) ) continue;
                   sum1 += ((float)(image1[index1])-mean1)*((float)(image2[index
2])-mean2);
                }

           sim = fabs(sum1/(cov1*cov2*num));

           return(sim);

}

/*********************************************************************
 INITIAL_MATCH - Initial matching

 Input - image1          first image
         width1          width of first image
         height1         height of first image
         f_pt1           feature points array
         num1            number of feature points
         match_pt        matched feature points array
         image2          second image
         width2          width of second image
         height2         height of second image
         f_pt2           feature points array
         num2            number of feature points

 Output - None

 Procedure calls - CORR_COEF()

 Return - 1 if success, otherwise -1

 Created by - Collins Chien on June 16th, 1994

*********************************************************************/
INITIAL_MATCH(image1, width1, height1, f_pt1, num1, match_pt, image2, width2, he
ight2, f_pt2, num2)
int *image1, *image2;
int width1, width2, height1, height2;
POINT *f_pt1, *f_pt2, *match_pt;
int num1, num2;
{
        int i, j, k, l;
        float max;
        int WS;
        int x1, y1, x2, y2;
        float val;
        float CORR_COEF();

        WS = 3;
        for(i=0; i<num1; i++) {
           x1 = f_pt1[i].x; y1 = f_pt1[i].y;
```

```
            max = 0.0; match_pt[i].x = -1; match_pt[1].mag = 0;
            for(j=0; j<num2; j++) {
                 for(k=-WS; k<=WS; k++)
                    for(l=-WS; l<=WS; l++) {
                         x2 = f_pt2[j].x+l; y2 = f_pt2[j].y+k;
                         if((x2 < 0) || (x2 > width2) || (y2 < 0) || (y2 > height
2) ) continue;
                         val = CORR_COEF(image1, width1, height1, x1, y1, image2,
 width2, height2, x2, y2);
                         if(val > max) {
                            match_pt[i].x = x2;
                            match_pt[i].y = y2;
                            match_pt[i].mag = val;
                            f_pt1[i].mag = val;
                            max = val;
                         }
                    }
            }
            if(match_pt[i].x == -1) {
                 f_pt1[i].x = -1;
                 f_pt1[i].mag = 0;
            }
        }
}

float CORR_COEF(image1, width1, height1, x1, y1, image2, width2, height2, x2, y2
)
int *image1, *image2;
int width1, height1, width2, height2;
int x1, y1, x2, y2;
{
        int i, j, index1, index2;
        float mean1, mean2, mean1_sq, mean2_sq;
        float cov1, cov2;
        float size2;
        float num, sum1, sum2;
        float sim;
        int WM;

        WM = 8;
           num = 0; sum1 = 0; sum2 = 0;
           for(i=-WM; i<=WM; i++)
                for(j=-WM; j<=WM; j++) {
                   index1 = (i+y1)*width1+j+x1;
                   index2 = (i+y2)*width2+j+x2;
                   if( ((i+y1)<0) || ((i+y1)>height1) || ((j+x1)<0) || ((j+x1)>w
idth1) || ((i+y2)<0) || ((i+y2)>height2) || ((j+x2)<0) || ((j+x2)>width2) || (im
age1[index1] < 0) || (image2[index2] < 0) ) continue;
                   sum1 += (float)image1[index1];
                   sum2 += (float)image2[index2];
                   num++;
                }
           mean1 = sum1/num;
           mean2 = sum2/num;
           sum1 = 0; sum2 = 0;
           mean1_sq = mean1*mean1;
           mean2_sq = mean2*mean2;
           for(i=-WM; i<=WM; i++)
                for(j=-WM; j<=WM; j++) {
                   index1 = (i+y1)*width1+j+x1;
                   index2 = (i+y2)*width2+j+x2;
                   if( ((i+y1)<0) || ((i+y1)>height1) || ((j+x1)<0) || ((j+x1)>w
idth1) || ((i+y2)<0) || ((i+y2)>height2) || ((j+x2)<0) || ((j+x2)>width2) || (im
age1[index1] < 0) || (image2[index2] < 0) ) continue;
                   sum1 += (float)(image1[index1]*image1[index1]) - mean1_sq;
                   sum2 += (float)(image2[index2]*image2[index2]) - mean2_sq;
                }
           cov1 = sqrt(fabs(sum1/(num-1)));
           cov2 = sqrt(fabs(sum2/(num-1)));
           sum1 = 0; sum2 = 0;
           for(i=-WM; i<=WM; i++)
                for(j=-WM; j<=WM; j++) {
                   index1 = (i+y1)*width1+j+x1;
                   index2 = (i+y2)*width2+j+x2;
                   if( ((i+y1)<0) || ((i+y1)>height1) || ((j+x1)<0) || ((j+x1)>w
idth1) || ((i+y2)<0) || ((i+y2)>height2) || ((j+x2)<0) || ((j+x2)>width2) || (im
age1[index1] < 0) || (image2[index2] < 0) ) continue;
                   sum1 += ((float)(image1[index1])-mean1)*((float)(image2[index
2])-mean2);
           }

           sim = fabs(sum1/(cov1*cov2*num));

           return(sim);

}

/*****************************************************************
 MERGE_LIST() - Merge two feature points array into one

 Input - f_pt1          feature point array 1
         match1_pt      matched feature point from array 1
         num1           number of points in the array
         f_pt2          feature point array 2
         match2_pt      matched feature point from array 2
         num2           number of points in the array
         matched_num    number of merged matched points

 Output - None

 Procedure calls - FIND_COR()

 Return - array of merged feature points in pair form
                matched[even].x ---> matched[odd].x
                matched[even].y ---> matched[odd].y

 Created by - Collins Chien on June 10th, 1994

*****************************************************************/
POINT *MERGE_LIST(f_pt1, match1_pt, num1, f_ptw, matchw_pt, numw, matched_num)
POINT *f_pt1, *match1_pt, *f_ptw, *matchw_pt;
int num1, numw, *matched_num;
{

       int i, j, index;
       int max_num, num;
```

```
        POINT *tmp_match, *matched;

        max_num = 2*(num1+numw);
        num = 0;
        tmp_match = (POINT *)calloc(max_num, sizeof(POINT));
        if(tmp_match == NULL) {
           fprintf(stderr, "Cannot allocate memory for tmp_match in MERGE_LIST()
\n");
           return(NULL);
        }

/*******************************
 Clean up multiply mapped points
*******************************/
        for(i=0; i<num1; i++)
           for(j=0; j<num1; j++) {
                if(f_pt1[i].x == -1)
                   continue;
                else if((i != j) && (match1_pt[i].x == match1_pt[j].x) && (match
1_pt[i].y == match1_pt[j].y)) {
                   if(match1_pt[i].mag > match1_pt[j].mag) {
                        f_pt1[j].x = -1;
                        match1_pt[j].mag = 0;
                   } else {
                        f_pt1[i].x = -1;
                        match1_pt[i].mag = 0;
                   }
                }
           }
        for(i=0; i<numw; i++)
           for(j=0; j<numw; j++) {
                if(f_ptw[i].x == -1)
                   continue;
                else if((i != j) && (matchw_pt[i].x == matchw_pt[j].x) && (match
w_pt[i].y == matchw_pt[j].y)) {
                   if(matchw_pt[i].mag > matchw_pt[j].mag) {
                        f_ptw[j].x = -1;
                        matchw_pt[j].mag = 0;
                   } else {
                        f_ptw[i].x = -1;
                        matchw_pt[i].mag = 0;
                   }
                }
           }
/*************************
 Merging two feature lists
*************************/
        for(i=0; i<num1; i++) {
           for(j=0; j<numw; j++) {
                if((match1_pt[i].x == f_ptw[j].x) && (match1_pt[i].y == f_ptw[j]
.y) && (f_pt1[i].x == matchw_pt[j].x) && (f_pt1[i].y == matchw_pt[j].y))
                   f_ptw[j].x = -1;
           }
           if(f_pt1[i].x == -1) continue;
           index = 2*num;
           tmp_match[index].x = f_pt1[i].x;
           tmp_match[index].y = f_pt1[i].y;
           tmp_match[index].mag = match1_pt[i].mag;
           tmp_match[index+1].x = match1_pt[i].x;
           tmp_match[index+1].y = match1_pt[i].y;
           tmp_match[index+1].mag = match1_pt[i].mag;
           num++;
        }
        for(i=0; i<numw; i++) {
           if(f_ptw[i].x == -1) continue;
           index = 2*num;
           tmp_match[index].x = matchw_pt[i].x;
           tmp_match[index].y = matchw_pt[i].y;
           tmp_match[index].mag = matchw_pt[i].mag;
           tmp_match[index+1].x = f_ptw[i].x;
           tmp_match[index+1].y = f_ptw[i].y;
           tmp_match[index+1].mag = matchw_pt[i].mag;
           num++;
        }
        num *= 2;
        matched = (POINT *)calloc(num, sizeof(POINT));
        if(matched == NULL) {
           fprintf(stderr, "Cannot allocate memory for matched in MERGE_LIST()\n
");
           return(NULL);
        }
        for(i=0; i<num; i++) {
           matched[i].x = tmp_match[i].x;
           matched[i].y = tmp_match[i].y;
           matched[i].mag = tmp_match[i].mag;
        }
        free(tmp_match);
        *matched_num = num;
        return(matched);
}

/* POINT *VERIFY(matched_pt, matched_num)
POINT *matched_pt;
int *matched_num;
{
        POINT *tmp1, *tmp2;
        int tmp_num, i, j, index;
        extern int verify();

        tmp_num = (*matched_num)/2;
        tmp1 = (POINT *)calloc(tmp_num, sizeof(POINT));
        tmp2 = (POINT *)calloc(tmp_num, sizeof(POINT));
        if((tmp1 == NULL) || (tmp2 == NULL)) {
           fprintf(stderr, "Cannot allocate memory for tmp1 and tmp2 in VERIFY()
\n");
           return(NULL);
        }
        for(i=0; i<tmp_num; i++) {
           index = 2*i;
           tmp1[i].x = matched_pt[index].x;
           tmp1[i].y = matched_pt[index].y;
           tmp1[i].mag = matched_pt[index].mag;
           tmp2[i].x = matched_pt[index+1].x;
           tmp2[i].y = matched_pt[index+1].y;
           tmp2[i].mag = matched_pt[index+1].mag;
        }
```

```
        verify(tmp1, tmp2, tmp_num);
        j = 0;
        for(i=0; i<tmp_num; i++)
           if(tmp1[i].mag > 0) j++;
        *matched_num = 2*j;
        free(matched_pt);
        matched_pt = (POINT *)calloc((*matched_num), sizeof(POINT));
        if(matched_pt == NULL) {
           fprintf(stderr, "Cannot allocate memory for matched_pt in VERIFY()\n"
);
           return(NULL);
        }
        j = 0;
        for(i=0; i<tmp_num; i++) {
           if(tmp1[i].mag > 0) {
                index = 2*j;
                matched_pt[index].x = tmp1[i].x;
                matched_pt[index].y = tmp1[i].y;
                matched_pt[index].mag = tmp1[i].mag;
                matched_pt[index+1].x = tmp2[i].x;
                matched_pt[index+1].y = tmp2[i].y;
                matched_pt[index+1].mag = tmp2[i].mag;
                j++;
           }
        }
        free(tmp1); free(tmp2);
        return(matched_pt);
}
*/

POINT *VERIFY(matched_pt, matched_num, level)
POINT *matched_pt;
int *matched_num;
int level;
{
        float corn[5][2], dx, dy;
        float WS;
        int i, j, num, num1, index, index1;
        float nom, denom, fxy_ratio;
        float s_b, c_b, s_g, c_g, s_a, c_a;
        float pan_angle_radian, tilt_angle_radian, rotate_angle_radian;
        extern float pan_angle, tilt_angle, rotate_angle;
        extern float focal_x, focal_y;
        extern float trans_x, trans_y, trans_z, depth;
        POINT *tmp_pt;
        int INSIDE();

        WS = 10;
        num = (*matched_num)/2;
        num1 = num;
        fxy_ratio = focal_x/focal_y;
        pan_angle_radian = pan_angle*DEGREE2RADIAN;
        tilt_angle_radian = tilt_angle*DEGREE2RADIAN;
        rotate_angle_radian = rotate_angle*DEGREE2RADIAN;
        s_b = (float)sin((double)(pan_angle_radian));
        c_b = (float)cos((double)(pan_angle_radian));
        s_g = (float)sin((double)(-tilt_angle_radian));
        c_g = (float)cos((double)(-tilt_angle_radian));
        s_a = (float)sin((double)(rotate_angle_radian));
        c_a = (float)cos((double)(rotate_angle_radian));
        s_a = 0.0;
        c_a = 1.0;
        corn[4][0] = 0;
        corn[4][1] = 0;
        nom = corn[4][0]*c_b*c_a+corn[4][1]*fxy_ratio*(c_a*s_b*s_g-s_a*c_g)+foca
l_x*(c_a*s_b*c_g+s_a*s_g)+focal_x*trans_x/(depth-trans_z);
        denom = -corn[4][0]*s_b/focal_x+corn[4][1]*c_b*s_g/focal_y+c_b*c_g+trans
_z/(depth-trans_z);
        dx = nom/denom;
        nom = corn[4][0]/fxy_ratio*s_a*c_b+corn[4][1]*(s_a*s_b*s_g+c_a*c_g)+foca
l_y*(s_a*s_b*c_g-c_a*s_g)+focal_y*trans_y/(depth-trans_z);
        dy = nom/denom;
        for(i=level; i>0; i--) {
           dx /= 2.0;
           dy /= 2.0;
        }
        for(i=0; i<num; i++) {
           index = 2*i;
           if(matched_pt[index].x == -1) {
                num1--;
                continue;
           }
           corn[0][0] = (float)(matched_pt[index].x)+WS-dx;
           corn[0][1] = (float)(matched_pt[index].y)-WS-dy;
           corn[1][0] = (float)(matched_pt[index].x)-WS-dx;
           corn[1][1] = (float)(matched_pt[index].y)-WS-dy;
           corn[2][0] = (float)(matched_pt[index].x)-WS-dx;
           corn[2][1] = (float)(matched_pt[index].y)+WS-dy;
           corn[3][0] = (float)(matched_pt[index].x)+WS-dx;
           corn[3][1] = (float)(matched_pt[index].y)+WS-dy;

           if(INSIDE(corn, matched_pt[index+1]) == -1) {
                matched_pt[index].x = -1;
                matched_pt[index].mag = 0;
                num1--;
           }
        }

/*******************************
 Clean up multiple mapped points
*******************************/
        for(i=0; i<num; i++) {
           index = 2*i;
           for(j=0; j<num; j++) {
                index1 = 2*j;
                if(matched_pt[index].x == -1)
                   continue;
                else if((i != j) && (matched_pt[index+1].x == matched_pt[index1+
1].x) && (matched_pt[index+1].y == matched_pt[index1+1].y)) {
                   if(matched_pt[index].mag > matched_pt[index1].mag) {
                        matched_pt[index1].x = -1;
                        matched_pt[index1].mag = 0;
                   } else {
                        matched_pt[index].x = -1;
                        matched_pt[index].mag = 0;
                   }
```

```
                num1--;
              }
           }
        }

        *matched_num = num1*2;
        tmp_pt = (POINT *)calloc((*matched_num), sizeof(POINT));
        if(tmp_pt == NULL) {
           fprintf(stderr, "Cannot allocate memory for tmp_pt in VERIFY()\n");
           return(NULL);
        }
        num1 = 0;
        for(i=0; i<num; i++) {
           index1 = 2*i;
           if(matched_pt[index1].x == -1) continue;
           index = 2*num1;
           tmp_pt[index].x = matched_pt[index1].x;
           tmp_pt[index].y = matched_pt[index1].y;
           tmp_pt[index].mag = matched_pt[index1].mag;
           tmp_pt[index+1].x = matched_pt[index1+1].x;
           tmp_pt[index+1].y = matched_pt[index1+1].y;
           tmp_pt[index+1].mag = matched_pt[index1+1].mag;
           num1++;
        }
        *matched_num = 2*num1;
        free(matched_pt);
        return(tmp_pt);
}

int INSIDE(c, m)
float c[5][2];
POINT *m;
{
        int xmin, xmax, ymin, ymax;

        xmin = (int)MIN(c[1][0], c[2][0]);
        ymin = (int)MIN(c[1][1], c[0][1]);
        xmax = (int)MAX(c[0][0], c[3][0]);
        ymax = (int)MAX(c[3][1], c[2][1]);

        if((xmin < m->x) && (m->x < xmax) && (ymin < m->y) && (m->y < ymax))
           return(1);
        else
           return(-1);
}

/*****************************************************************
 ESTIMATE_SCALE() - Estimate the scaling factor

 Input - matched_pt     matched feature points array
         matched_num    number of matched feature points

 Output - None

 Procedure calls - None

 Return - scaling factor, if fail, -1

 Created by - Collins Chien on June 10th, 1994

*****************************************************************/
float ESTIMATE_SCALE(matched_pt, matched_num)
POINT *matched_pt;
int matched_num;
{

        int sum1x, sum1y, sum2x, sum2y;
        float mean1x, mean1y, mean2x, mean2y;
        int num, i, index;
        float *D1, *D2;
        float x, y, nom, den;

        num = matched_num/2;
        sum1x = 0; sum1y = 0; sum2x = 0; sum2y = 0;
        for(i=0; i<num; i++) {
           index = 2*i;
           sum1x += matched_pt[index].x;
           sum1y += matched_pt[index].y;
           sum2x += matched_pt[index+1].x;
           sum2y += matched_pt[index+1].y;
        }
        mean1x = (float)sum1x/(float)num;
        mean1y = (float)sum1y/(float)num;
        mean2x = (float)sum2x/(float)num;
        mean2y = (float)sum2y/(float)num;

        D1 = (float *)calloc(num, sizeof(float));
        D2 = (float *)calloc(num, sizeof(float));
        if((D1 == NULL) || (D2 == NULL)) {
           fprintf(stderr, "Cannot allocate D1 and D2 in ESTIMATE_SCALE()\n");
           free(D1);
           free(D2);
           return(-1);
        }
        for(i=0; i<num; i++) {
           index = 2*i;
           x = (float)(matched_pt[index].x) - mean1x;
           y = (float)(matched_pt[index].y) - mean1y;
           D1[i] = hypot(x, y);
           x = (float)(matched_pt[index+1].x) - mean2x;
           y = (float)(matched_pt[index+1].y) - mean2y;
           D2[i] = hypot(x, y);
        }
        nom = 0; den = 0;
        for(i=0; i<num; i++) {
           nom += (D1[i]*D2[i]);
           den += (D1[i]*D1[i]);
        }

        free(D1);
        free(D2);
        return(nom/den);
}

/*****************************************************************
 UPDATE_PARAMETERS() - update rotational angle, scaling factor
```

```
                    horizontal and vertical translation parameters

 Input - matched_pt     matched feature points array
         matched_num    number of matched feature points
         theta          rotational angle
         scale          scaling factor
         dx             horizontal translation
         dy             vertical translation

 Output - None

 Procedure calls - ESTIMATE_SCALE()
                   ESTIMATE_ROT_TRANS()

 Return - 1 if success, otherwise, -1

 Created by - Collins Chien on June 10th, 1994

*******************************************************************/
UPDATE_PARAMETERS(matched_pt, matched_num, theta, scale, dx, dy)
POINT *matched_pt;
int matched_num;
float *theta, *scale;
int *dx, *dy;
{
        float ESTIMATE_SCALE();
        int ESTIMATE_ROT_TRANS();
        float delta_scale, delta_theta;
        int delta_dx, delta_dy;
        float s_t, c_t, t;

        delta_scale = ESTIMATE_SCALE(matched_pt, matched_num);
        if(delta_scale == -1)
           return(-1);

        if(ESTIMATE_ROT_TRANS(matched_pt, matched_num, delta_scale, &delta_theta
, &delta_dx, &delta_dy) == -1)
           return(-1);

        *scale = (*scale)*delta_scale;
        *theta = (*theta)+delta_theta;
        t = delta_theta;
        s_t = sin(t);
        c_t = cos(t);
        *dx = delta_scale*(c_t*(*dx)+s_t*(*dy))+delta_dx;
        *dy = delta_scale*(-s_t*(*dx)+c_t*(*dy))+delta_dy;
}

/*******************************************************************
 ESTIMATE_ROT_TRANS() - estimate rotational angle, horizontal and
                        vertical translation parameters

 Input - matched_pt     matched feature points array
         matched_num    number of matched feature points
         scale          scaling factor
         theta          rotational angle
         dx             horizontal translation
         dy             vertical translation

 Output - None

 Procedure calls - sqrls_()

 Return - 1 if success, otherwise, -1

 Created by - Collins Chien on June 10th, 1994

*******************************************************************/
int ESTIMATE_ROT_TRANS(matched_pt, matched_num, scale, theta, dx, dy)
POINT *matched_pt;
int matched_num;
float scale, *theta;
int *dx, *dy;
{

        extern int sqrls_();
        float *a, *b, *qraux, *work, *c;
        int *jpvt, kr, tol, ma, ndata, itask, ind;
        float dum1, dum2, num;
        int i, index;


/*****************************************
 Create and build matrices for least square
*****************************************/
        ndata = matched_num;
        ma = 3;
        a = (float *)calloc(ndata*ma, sizeof(float));
        if( a == NULL ) {
           fprintf(stderr, "Unable to create memory for a\n");
           return(-1);
        }
        b = (float *)calloc(ndata, sizeof(float));
        if( b == NULL ) {
           fprintf(stderr, "Unable to create memory for b\n");
           free(a);
           return(-1);
        }
        qraux = (float *)calloc(ma, sizeof(float));
        if( qraux == NULL ) {
           fprintf(stderr, "Unable to create memory for qraux\n");
           free(a);
           free(b);
           return(-1);
        }
        jpvt = (int *)calloc(ma, sizeof(int));
        if( jpvt == NULL ) {
           fprintf(stderr, "Unable to create memory for jpvt\n");
           free(a);
           free(b);
           free(qraux);
           return(-1);
        }
        work = (float *)calloc(ma, sizeof(float));
        if( work == NULL ) {
           fprintf(stderr, "Unable to create memory for work\n");
```

```
            free(a);
            free(b);
            free(qraux);
            free(jpvt);
            return(-1);
         }
         c = (float *)calloc(ma, sizeof(float));
         if( c == NULL ) {
            fprintf(stderr, "Unable to create memory for c\n");
            free(a);
            free(b);
            free(qraux);
            free(jpvt);
            free(work);
            return(-1);
         }
         num = ndata/2;
         for(i=0; i<num; i++) {
            index = 2*i;
            dum1 = scale*(float)(matched_pt[index].x);
            dum2 = scale*(float)(matched_pt[index].y);
            a[index] = dum2;
            a[ndata+index] = 1.0;
            a[ndata*2+index] = 0.0;
            a[index+1] = -dum1;
            a[ndata+index+1] = 0.0;
            a[ndata*2+index+1] = 1.0;
            b[index] = (float)(matched_pt[index+1].x)-dum1;
            b[index+1] = (float)(matched_pt[index+1].y)-dum2;
         }
         tol = 1.0E-6;
         itask = 1;

/*****************************************
 Call sqrls to do least square calculation
*****************************************/
         sqrls_(a, &ndata, &ndata, &ma, &tol, &kr, b, c, b, jpvt, qraux, work, &i
task, &ind);

         *theta = c[0];
         *dx = nint(c[1]);
         *dy = nint(c[2]);

         free(a);
         free(b);
         free(c);
         free(qraux);
         free(jpvt);
         free(work);

         return(1);
}

/*****************************************************************
 MAGNIFY_FP() - magnify feature points coordinates one level higher

 Input - matched_pt     matched feature points array
         matched_num    number of matched feature points

 Output - None

 Procedure calls - None

 Return - 1 if success, otherwise, -1

 Created by - Collins Chien on June 11th, 1994

*****************************************************************/
MAGNIFY_FP(matched_pt, matched_num)
POINT *matched_pt;
int matched_num;
{
         int i;

         if(matched_pt == NULL)
            return(-1);
         for(i=0; i<matched_num; i++) {
            matched_pt[i].x *= 2.0;
            matched_pt[i].y *= 2.0;
         }
         return(1);
}
```

```
#include <stdio.h>
#include <math.h>
#include "../include/struct.h"

/*****************************************************************************
 find_coef - calculate coefficients of the mapping function

 Description -
        Given the corner coordinates of the original image (u, v) and the warped
        image (x, y), we can form a system of linear equation.

                |u_1|   |1 x_1 y_1 x_1y_1| |a_1|
                |u_2|   |1 x_2 y_2 x_2y_2| |a_2|
                |u_3| = |1 x_3 y_3 x_3y_3| |a_3|
                |u_4|   |1 x_4 y_4 x_4y_4| |a_4|

        The same system of equation for v and b.
        This program us LU decomposition method to solve for a's and b's.
        The algorithm is from Linpack software package.  The original code is
        written in FORTRAN but I translated it into C by the tool f2c.  As
        a result, you have to compile it with the f2c library.  When you
        are compiling, remember to put the following items in your makefile:
                1. toollib.a
                2. libf2c.a

        toollib.a is located at ~collins/Thesis/Tools/bin/sun4bin and
        libf2c.a is in /usr/gnu/lib

 Input -
        org_pt          corner points of the original image and the order is
                        starting from top-right corner and go anti-clockwise
        warp_pt         corner points of the warped image with the same order
                        as previous
        n               number of element in the corner arrays
        c               output coefficients
                        c[0] - c[3] => a[0] - a[3]
                        c[4] - c[7] => b[0] - b[3]

        Note that both org_pt and warp_pt are in POINT type.

 Procedure called -
        sgefs_()        Linpack function

 Output -
        Return -1 when error occur, 1 otherwise

 Author -
        Collins Chien, August 4, 1994.

*****************************************************************************/
find_coef(org_pt, warp_pt, n, c)
POINT   *org_pt;
POINT   *warp_pt;
int     n;
float   *c;
{
        float *a, *b, *work, rcond;
        int *iwork, itask, ind;
        int i, j;

        a = (float *)calloc(n*n, sizeof(float));
        if(a == NULL) {
           fprintf(stderr, "Cannot allocate memory for a in warp_image()\n");
           return(-1);
        }
        b = (float *)calloc(n, sizeof(float));
        if(b == NULL) {
           fprintf(stderr, "Cannot allocate memory for b in warp_image()\n");
           free(a);
           return(-1);
        }
        work = (float *)calloc(n, sizeof(float));
        if(work == NULL) {
           fprintf(stderr, "Cannot allocate memory for work in warp_image()\n");
           free(a);
           free(b);
           return(-1);
        }
        iwork = (int *)calloc(n, sizeof(int));
        if(iwork == NULL) {
           fprintf(stderr, "Cannot allocate memory for iwork in warp_image()\n")
;
           free(a);
           free(b);
           free(work);
           return(-1);
        }
/*************************************
 Initialize work, iwork, itask and rcond
*************************************/
        for(i=0; i<n; i++) {
           work[i] = 0.0;
           iwork[i] = 0;
        }
        itask = 1;
        rcond = 0;
        ind = 0;
/**************
 Form A matrix
***************/
        for(i=0; i<n; i++) {
           a[i] = 1.0;
           a[i+n] = warp_pt[i].x;
           a[i+2*n] = warp_pt[i].y;
           a[i+3*n] = (warp_pt[i].x) * (warp_pt[i].y);
        }
/**************
 Form B matrix
***************/
        for(i=0; i<n; i++)
           b[i] = org_pt[i].x;
/*************
 Call sgefs()
**************/
        sgefs_(a, &n, &n, b, &itask, &ind, work, iwork, &rcond);
/******************
```

```
 Store coefficients
*********************/
        for(j=0; j<n; j++)
           c[j] = b[j];
/**************
 Form B matrix
**************/
        for(i=0; i<n; i++)
           b[i] = org_pt[i].y;
/*****************************************************
 Set flag to indicate no matrix factorization is needed
*****************************************************/
        itask = 2;
/*************
 Call sgefs()
*************/
        sgefs_(a, &n, &n, b, &itask, &ind, work, iwork, &rcond);
/*******************
 Store coefficients
*******************/
        for(j=n; j<n*2; j++)
           c[j] = b[j-n];
/*********************************
 Print coefficients and error code
*********************************/
        fprintf(stdout, "ind = %d\n", ind);
        for(i=0; i<2*n; i++)
           fprintf(stdout, "c[%d] = %.31f\n", i, c[i]);
        return(1);
}

/*****************************************************************************
 warp_image - warp an image into another shape

 Description -
        Given the original image A and the desired warped point positions, this
        program uses backward mapping technique to find the pixel coordinates
        and uses bilinear interpolation method to find the greylevel value.
        Let (u,v) be the coordinates of the original image A and (x,y) be the
        warped image B.  Backward mapping is

                        u = a_1 + a_2x + a_3y + a_4xy
                        v = b_1 + b_2x + b_3y + b_4xy

        After we obtained the coefficients of these polynomial, we use
        bilinear interpolation

                        f_u = floor(u)
                        f_v = floor(v)
                        alpha = u - f_u
                        beta = v - f_v

        The greylevel image B at (x,y) is

        B[x,y] = (1-alpha)*(1-beta)*A[f_u,f_v] + (1-alpha)*beta*A[f_u,f_v+1]
                + alpha*(1-beta)*A[f_u+1,f_v] + alpha*beta*A[f_u+1,f_v+1];

        When you are compiling, remember to put the following items in your
        makefile:
                1. toollib.a
                2. libf2c.a

        toollib.a is located at ~collins/Thesis/Tools/bin/sun4bin and
        libf2c.a is in /usr/gnu/lib

 Input -
        org_image       origial image
        width           width of original image
        height          height of original image
        warp_image      warped image
        w_width         width of warped image
        w_height        height of warped image
        warp_pt         warped image corners and the order is starting from
                        the top-right corner and go anti-clockwise
        init_val        initial value of the warped image - they can be the
                        values of the blank area after warping

        Note that warp_pt are in POINT type.  You must define it somewhere in
        your calling program.  It is like this:

                typedef struct pt
                {
                   int x, y;
                } POINT;

 Procedure called -
        find_coef()     calculate polynomial coefficients

 Output -
        Return -1 when error occur, 1 otherwise

 Author -
        Collins Chien, August 4, 1994.

*****************************************************************************/
int warp_image(org_image, width, height, warp_image, w_width, w_height, warp_pt,
 init_val)
int *org_image;
int width, height;
int *warp_image;
int w_width, w_height;
POINT *warp_pt;
int init_val;
{

        POINT org_pt[4];
        int i, j, index, n;
        float coef[8];
        float t1, t2, x, y;
        float alpha, beta;
        float f_x, f_y;

/***************
 Initiailization
```

```
****************/
        n = 4;
        for(i=0; i<n; i++) {
           org_pt[i].mag = 0.0;
           org_pt[i].index = 0;
           coef[i] = 0.0; coef[n+i] = 0.0;
        }
/*****************************************************************************
***
 Assign org_pt in anti-clockwise order starting from top-right corner of the ima
ge
*****************************************************************************
***/
        org_pt[0].x = width-1; org_pt[0].y = 0;
        org_pt[1].x = 0;       org_pt[1].y = 0;
        org_pt[2].x = 0;       org_pt[2].y = height-1;
        org_pt[3].x = width-1; org_pt[3].y = height-1;
/*************************************************
 Find coefficients of the transformation function
*************************************************/
        find_coef(org_pt, warp_pt, n, coef);
        for(i=0; i<n; i++) {
           t1 = coef[0] + coef[1]*(warp_pt[i].x) + coef[2]*(warp_pt[i].y) + coef
[3]*(warp_pt[i].x)*(warp_pt[i].y);
           t2 = coef[4] + coef[5]*(warp_pt[i].x) + coef[6]*(warp_pt[i].y) + coef
[7]*(warp_pt[i].x)*(warp_pt[i].y);
           fprintf(stdout, "(%d, %d) -> (%.3lf, %.3lf)\n", warp_pt[i].x, warp_pt
[i].y, t1, t2);
        }
        for(i=0; i< w_height; i++)
           for(j=0; j< w_width; j++) {
                index = i*w_width+j;
                warp_image[index] = init_val;
                x = coef[0] + coef[1]*(float)j + coef[2]*(float)i + coef[3]*(flo
at)(i*j);
                y = coef[4] + coef[5]*(float)j + coef[6]*(float)i + coef[7]*(flo
at)(i*j);
                if((x>=0) && (x<(float)width) && (y>=0) && (y<(float)height)) {
                   f_x = floor(x);
                   f_y = floor(y);
                   alpha = x - f_x;
                   beta = y - f_y;

                   if((f_x < (float)(width-1)) && (f_y < (float)(height-1))) {
                        if( ((alpha == 0) && (beta == 0)) || ((alpha == 1) && (b
eta == 1)) || ((alpha == 0) && (beta == 1)) || ((alpha == 1) && (beta == 0)))
                           warp_image[index] = (org_image[(int)y*width+(int)x]+o
rg_image[(int)y*width+(int)(x+1)]+org_image[(int)(y+1)*width+(int)x]+org_image[(
int)(y+1)*width+(int)(x+1)])/4;
                        else
                           warp_image[index] = (1-alpha)*(1-beta)*org_image[(int
)f_y*width+(int)f_x] + (1-alpha)*beta*org_image[((int)f_y+1)*width+(int)f_x] + a
lpha*(1-beta)*org_image[(int)f_y*width+(int)f_x+1] + alpha*beta*org_image[((int)
f_y+1)*width+(int)f_x+1];
                   } else if(f_x < (float)(width-1)) { /* boundary condition */
                        if( ((alpha == 0) && (beta == 0)) || ((alpha == 1) && (b
eta == 1)) || ((alpha == 0) && (beta == 1)) || ((alpha == 1) && (beta == 0)))
                           warp_image[index] = (org_image[(int)y*width+(int)x]+o
rg_image[(int)y*width+(int)(x+1)])/2;
                        else
                           warp_image[index] = (1-alpha)*org_image[(int)f_y*widt
h+(int)f_x] + alpha*org_image[(int)f_y*width+(int)f_x+1];
                   } else if(f_y < (float)(height-1)) { /* boundary condition */
                        if( ((alpha == 0) && (beta == 0)) || ((alpha == 1) && (b
eta == 1)) || ((alpha == 0) && (beta == 1)) || ((alpha == 1) && (beta == 0)))
                           warp_image[index] = (org_image[(int)y*width+(int)x]+o
rg_image[(int)(y+1)*width+(int)x])/2;
                        else
                           warp_image[index] = (1-beta)*org_image[(int)f_y*width
+(int)f_x] + beta*org_image[((int)f_y+1)*width+(int)f_x];
                   } else
                        warp_image[index] = org_image[(int)f_y*width+(int)f_x];

                   if(warp_image[index] > 255)
                        warp_image[index] = 255;
                   else if(warp_image[index] < 0)
                        warp_image[index] = -1;
                }
           }
        return(1);
}

int *scale_image(org_image, width, height, factor)
int *org_image;
int width, height;
float factor;
{
        int warp_image();
        POINT red_pt[4];
        int *red_image,
        int red_width, red_height;

        red_width = nint((float)width*factor);
        red_height = nint((float)height*factor);

        red_image = (int *)calloc(red_width*red_height, sizeof(int));
        if(red_image == NULL) {
           fprintf(stderr, "Cannot allocate memory to red_image in scale_image()
\n");
           return(NULL);
        }
/*****************************************************************************
 Assign four corners in anticlockwise direction starting from top-right corner
*****************************************************************************/
        red_pt[0].x = red_width-1;red_pt[0].y = 0;
        red_pt[1].x = 0;          red_pt[1].y = 0;
        red_pt[2].x = 0;          red_pt[2].y = red_height-1;
        red_pt[3].x = red_width-1;red_pt[3].y = red_height-1;
        if(warp_image(org_image, width, height, red_image, red_width, red_height
, red_pt))
           return(red_image);
}
```

```
#include <stdio.h>
#include <math.h>
#include <rasterfile.h>
#include "../include/struct.h"
#include "../include/constants.h"

/*****************************************************************************
 sm2() - Similarity measure method using Sum of Absolute Valued Differences

 Input - aImage          : XImage structure with the actual image
         LeftTopX        : x coordinate of left top corner of bounding box
         LeftTopY        : y coordinate of left top corner of bounding box
         RightBottomX    : x coordinate of right bottom corner of bounding box
         RightBottomY    : y coordinate of right bottom corner of bounding box
         aTemplate       : Template data
         Twidth          : Template width
         Theight         : Template height

Process - It crop out the region defined by LeftTopX, LeftTopY, RightBottomX
          and RightBottomY first and performs Sum of Absolute Valued
          Differences on that region.  Then report the best matched position.

Output - None.

Return value - When fail to allocate memory, it returns -1.
               If it successfully processes the image, it returns 1.

Note - Remember to include 'stdio.h' and 'X11/Xlib.h' when using this module.

Created by - Collins Chien on 22nd July, 1994.
             Modified to corner detector on 22nd November, 1993.

*****************************************************************************/
sm2(aImage, image_width, LeftTopX, LeftTopY, RightBottomX, RightBottomY, aTempla
te, Twidth, Theight)
int     *aImage;                /* The image */
int     image_width;
int     LeftTopX, LeftTopY;     /* Left top corner of the bounding box */
int     RightBottomX, RightBottomY;     /* Right bottom corner of the bounding b
ox */
int     *aTemplate;
int     Twidth, Theight;
{
        int     *temp_mem;
        int     *pixel_ptr;
        int     i, j, k, index;
        int     m, mm, index1, bound1, bound2, bound3, *pos, total;
        int     box_width, box_height;
        float   mag;
        float   max_mag, min_mag;
        POINT   m_pt;
        float   F_SAVD();

        box_width = RightBottomX - LeftTopX;
        box_height = RightBottomY - LeftTopY;
        total = box_width*box_height;

        if(box_width <= 0 || box_height <= 0) {
           fprintf(stderr, "Invalid bounding box!\n");
           return(-1);
        }
        temp_mem = (int *)calloc(box_width*box_height, sizeof(int));
        if(temp_mem == NULL) {
           fprintf(stderr, "Unable to create memory for the bounded image!\n");
           return(-1);
        }

/*************************************************************
 Copy bounding box image into temporary memory for processing
*************************************************************/
        pixel_ptr = aImage;
        for(i=0; i<box_height; i++) {
           for(j=0; j<box_width; j++) {
                index = image_width*(i+LeftTopY) + j+LeftTopX;
                temp_mem[i*box_width+j] = pixel_ptr[index];
           }
        }

        max_mag = -1000000;
        min_mag = 1000000;

        for(i=0; i<box_height-Theight; i++)
           for(j=0; j<box_width-Twidth; j++) {
                mag = F_SAVD(temp_mem, box_width, box_height, j, i, aTemplate, T
width, Theight);
                if(mag < min_mag) {
                   m_pt.x = LeftTopX+j;
                   m_pt.y = LeftTopY+i;
                   m_pt.mag = mag;
                   min_mag = mag;
                }
           } /* for loop */


        fprintf(stdout, "Best match is at (%d, %d) with mag= %.3lf\n",m_pt.x, m_
pt.y, m_pt.mag);


        free(temp_mem);
        return(1);
}

/*************************************************************
 F_SAVD() - Find sum of absolute valued differences (SAVD)

 Input - image1          Image
         width1          width of the image
         height1         height of the image
         x               x coordinate of the potential match point
         y               y coordinate of the potential match point
         image2          Template
         width2          width of template
         height2         height of template

 Output - None
```

```
 Procedure calls - None

 Return - correlation value, -1 if fail to create memory

 Created by - Collins Chien on July 22nd, 1994

*****************************************************************/
float F_SAVD(image1, width1, height1, x, y, image2, width2, height2)
int *image1, *image2;
int width1, width2;
int height1, height2;
int x, y;
{

        int i, j, index1, index2;
        float sim;

        sim = 0;
        for(i=0; i<height2; i++)
                for(j=0; j<width2; j++) {
                   index1 = (i+y)*width1+j+x;
                   index2 = i*width2+j;
                   if( ((i+y)<0) || ((i+y)>height1) || ((j+x)<0) || ((j+x)>width
1) || (image1[index1] < 0) || (image2[index2] < 0) ) continue;
                   sim += (float)abs(image1[index1] - image2[index2]);
                }

        return(sim);
}
```

```
#include "../include/constants.h"
#include "../include/struct.h"

/* This draw_cross() is for grey level image */
draw_cross(f, t, i, w, s)
int *f;
int *t;
int i, w, s;
{
        int j;

        for(j=-s; j<=s; j++) {
           if(t[i+j] > 127)
               f[i+j] = BLACK;
           else
                f[i+j] = WHITE;
           if(j != 0) {
                if(t[i+j*w] > 127)
                   f[i+j*w] = BLACK;
           else
                   f[i+j*w] = WHITE;
           }
        }
}

int fcmp(a, b)
POINT *a, *b;
{
/***************
 Decending order
****************/
        if(a->mag > b->mag) return(-1);
        if(a->mag < b->mag) return(1);
        return(0);
}

int calc_mag(image, w, pt, num, side)
int *image, num, w, side;
POINT *pt;
{
        int i, j, k, index, half_side;

        half_side = side/2;

        for(i=0; i<num; i++) {
           index = pt[i].index;
           if(pt[i].mag > 0) {
                pt[i].mag = 0;
                for(j=-half_side; j<=half_side; j++)
                   for(k=-half_side; k<=half_side; k++)
                        pt[i].mag += image[index+j*w+k];
           }
        }
}

/*************************************************************
 *
 *   Function name : DrawArrow
 *
 *   Description :
 *      Draw a line between two points by using Bresenham's
 *      integer line drawing algorithm.  The end of the line
 *      is indicated by a 10*10 pixels square.
 *
 *   Inputs :
 *      int *img                - input image
 *      int w                   - image width
 *      int xx1, yy1            - starting point
 *      int xx2, yy2            - ending point
 *
 *   Outputs :
 *     Draw a straight line in img
 *
 *   Returned Value : none
 * -----------------------------------------------------------
 *
 *   Original : Collins Chien
 *   Modified : From BYTE_IMAGE to integer image (May 6th, 1994)
 *
 *************************************************************/

DrawArrow(img, w, xx1, yy1, xx2, yy2)
int *img, w;
int xx1, yy1, xx2, yy2;
{
  int dx, dy;           /* x and y differential */
  int e_inc;            /* change in error when y increments */
  int e_noinc;          /* change in error when no increment in y */
  int e;
  int flag=0;
  int x=xx1, y=yy1;
  int x1=xx1, y1=yy1, x2=xx2, y2=yy2;
  int incr;
  int i, j;
  double m;
  void sswap();
  int set_pixel();

  dx=x2-x1;
  dy=y2-y1;
  if( dx==0 ) {         /* vertical line */
    if( dy <= 0 ) {
        sswap( &y1, &y2 );
        dy=-dy;
    }
    for( y=y1; y<=y2; y++ )
        set_pixel( img, w, x1, y );
  } else if( dy==0 ) {
    if( dx <= 0 ) {     /* horizontal line */
        sswap( &x1, &x2 );
        dx=-dx;
    }
    for( x=x1; x<=x2; x++ )
        set_pixel( img, w, x, y1 );
  }  else {
    m=(double)dy/(double)dx;
```

```
    if( x2 < x1 ) {
        sswap( &x1, &x2 );
        sswap( &y1, &y2 );
    }
    if(( m>1 )||( m<-1 )) {      /* line in section 2 or 3 */
        sswap( &x1, &y1 );       /* flip the line into section 1 or 4 */
        sswap( &x2, &y2 );
        flag=1;
        if( x2<x1 ) {
          sswap( &x1, &x2 );
          sswap( &y1, &y2 );
        }
    }
    dx=x2-x1;                    /* Bresenham's algorithm */
    dy=y2-y1;
    if( m>0 )
        incr=1;
    else {
        dy=-dy;
        incr=-1;
    }
    e_noinc=dy+dy;
    e=e_noinc-dx;
    e_inc=e-dx;
    x=0;
    y=0;
    for( x=0; x<=dx; x++ ) {
        if( flag )               /* flip back x and y coordinates */
          set_pixel( img, w, y+y1, x+x1 );
        else
          set_pixel( img, w, x+x1, y+y1 );
        if( e<0 )
          e+=e_noinc;
        else {
          y+=incr;
          e+=e_inc;
        }
    }
  }
  for(i=0; i<6; i++)             /* Drawing a square at the end point */
     for(j=0; j<6; j++)
        set_pixel(img, w, xx2-5+i, yy2-5+j);
}

/************************************************************
 *
 *   Function name : sswap
 *
 *   Description :
 *      Swap the values between two variables
 *
 *   Inputs :
 *      int a, b        - variables to be swapped
 *
 *   Outputs :
 *     none.
 *
 *   Returned Value : none
 * ----------------------------------------------------------
 *
 *   Original : Collins Chien
 *   Modified :
 *
 ************************************************************/

void sswap(a, b)  /* This is a simple swap routine which */
int *a, *b;       /* swap the value of a and b */
{
  register int temp;
  temp=*a;
  *a=*b;
  *b=temp;
}

/************************************************************
 *
 *   Function name : set_pixel
 *
 *   Description :
 *      Turn on the specified pixel to white
 *
 *   Inputs :
 *      int *img                - input image
 *      int w                   - image width
 *      int x, y                - coordinate of the pixel
 *
 *   Outputs :
 *     none.
 *
 *   Returned Value : none.
 * ----------------------------------------------------------
 *
 *   Original : Collins Chien
 *   Modified : From BYTE_IMAGE to integer image (May 6th, 1994)
 *
 ************************************************************/

set_pixel(img, w, x, y)
int *img;
int w;
int x, y;
{
        int index;

        index = y*w+x;
        if(img[index] > 127)
           img[index] = BLACK;
        else
           img[index] = WHITE;
}
```