MODEL-BASED VISUAL RECOGNITION OF 3-D OBJECTS USING PSEUDO-RANDOM GRID ENCODING

by

NICULAIE TRIF

MAY 1993

A thesis submitted to the
School of Graduate Studies and Research
in partial fulfilment of the
requirements for the degree of
Master of Applied Sciences
in Electrical Engineering

OTTAWA-CARLETON INSTITUTE FOR ELECTRICAL ENGINEERING

Department of Electrical Engineering
Faculty of Engineering

University of Ottawa, Ontario, CANADA

ISBN 0-315-89718-X

Canada

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

I hereby declare that I am the sole author of this thesis.

I authorize the University of Ottawa to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Niculaie Trif

I further authorize the University of Ottawa to reproduce this thesis by photocopying or by any other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Niculaie Trif

CONTENTS

APPENDIX

LIST OF FIGURES

LIST OF TABLES

LIST OF TERMS

PRS	Pseudo-Random Sequence
PRA	Pseudo-Random Array
PN	Pseudo-Noise
POSE	Position, Orientation, and Scale Estimation
AGV	Automated Guided Vehicle
$GF(q=p^m)$	Galois Field of p^m elements
DDL	Data Definition Language
DML	Data Manipulation Language
DBMS	Database Management System
PRW	Pseudo-Random Window
P4P	Perspective-4-Points
PRBS	Pseudo-Random Binary Sequence
PRBA	Pseudo-Random Binary Array

To my parents...

ACKNOWLEDGEMENTS

First, I would like to thank Dr. Emil Petriu, my thesis supervisor, for his direction, guidance and advice. His time and patience to communicate his ideas, knowledge, judgement, and experience have been rewarded with the results of the research presented in this thesis.

Next, I wish to thank the Canadian Space Agency for providing excellent working conditions and access to computer facilities to carry out the work reported in this thesis. I would like to thank Dr. William McMath for providing advice and encouragement. I would also like to thank Mr. Steven Yeung for clarifying to me some problems encountered during the vision system integration .

Next, I would like to thank Dr. Dorina Petriu for her recommendations and reviewing of the relational database solution to my thesis.

Finally, I would like to thank Mr. Jagdeep S. Basran for reviewing my thesis and for his good advice.

ABSTRACT

This thesis presents a solution to the following problem:

> *Given* a set of 3-D objects having their visible surfaces marked with symbols representing the terms of a pseudo-random array (PRA), *given* the mapping between the planar faces of the polyhedral models of these objects, and *given* a monocular (2-D) image partly showing an object of this set, determine object identity and its POSE (Position, Orientation and Scale Estimation) parameters relative to the 3-D frame of the video-camera.

From the symbols which could be recognized in the image, a broken portion of the PRA is reconstructed and then inspected until a complete pseudo-random window (PRW) is found. Such a window has the uniqueness property which allows the full identification of the window's absolute coordinates (i,j) within the PRA. By consulting the "object model/ PRA" mapping database it is possible to identify the object and its specific face containing the recovered window. Once the object face is identified, its POSE parameters are calculated from the perspective transformations relating the 2-D positions of the recognized symbols in the image frame to their correspondent PRA grid nodes defined in the 3-D object model frame (a classical "camera calibration" problem.)

PART 1

INTRODUCTION

The new industrial revolution that is in the making could be the result of the 'intelligent connection of the computer perception to robotic action' [38]. The only way to improve the perception capabilities of intelligent robots is to use sensors that allow them to efficiently adapt to changing environmental conditions.

Robots that are deaf and dumb or that have no sense of force or touch perform only manually trainable tasks that can be dangerous in some working environments. Many applications call for an "intelligent" robot, a stand-alone machine, usually with its own visual, contact, or auditory sensory perception system, that can detect changes in the three dimensional (3-D) working environment and adapt to them [1].

Two types of sensors are used in robotic systems: (i) *proprioceptors* for the measurement of the robot's (internal) parameters, and (ii) *exteroceptors* for the measurement of the environmental (external, from the robot's point of view) parameters.

Proprioceptors are sensors measuring both kinematic and dynamic parameters of the robot. The usual kinematic parameters are the joint positions, velocities, and accelerations. Dynamic parameters such as forces and torques are also important to measure for the proper control of robot arms.

Exteroceptors are sensors which measure the position and/or the force-type interaction of the robot with its environment. They can be classified according to their range as follows:

- *Contact sensors* detect the contact between the mating parts and/or measure the interaction forces and torques occurring in assembly operation. Another type of sensors are the tactile sensors which measure the contact force profile, slippage, or object shape.
- *Proximity sensors* detect objects which are near but without touching them.
- *'Far away' sensors*, which are of two types: range sensors which measure the distance to objects, and visual sensors such as cameras. They have proved to be the most important kinds of sensors in use.

1.1. 3-D Object Recognition

The recognition of three-dimensional objects, using visual sensors, is one of the most important tasks in robotics that has to be solved to make this revolution a reality.

The 3-D Object Recognition Paradigm

The 3-D arbitrary-view object recognition paradigm was clearly defined by Besel and Jain in their review paper [7] as follows:

Given

- any collection of labelled solid objects in which
  - each object may be examined as long as the object is not deformed
  - labelled models may be created using information from this examination

- an image corresponding to one particular, but arbitrary, field of view of the real world
- a database of previously defined (modelled) objects

Determine

- what object appears in a given image
- the 3-D location and orientation of the object with respect to a known coordinate system

Figure 1.1 shows the general 3-D object-recognition framework in which the object recognition system can be discussed. The fundamental domains of the system are:

- *The real world domain* composed of solid 3-D objects that the system has to recognize
- *The digitized sensor data domain* contains grey-scale intensity images or digitized range images of the objects that have to be recognized
- *The modelling domain* is composed of all the information that the system has about the real world. View dependent or view independent techniques can be used, but because representation is such a critical factor for any object-recognition system, the view independent world model is preferred
- *The intermediate symbolic scene description domain* is an intermediary domain that makes possible the matching between the sensor data and the model data.

The mapping between these domains are:

- *The image-formation process* (I) produces grey-scale intensity images or digitized range images
- *The description process* (D) acts on the sensor data and extracts relevant object features. This process should be application independent and incorporate only the knowledge of the image formation process and rudimentary facts

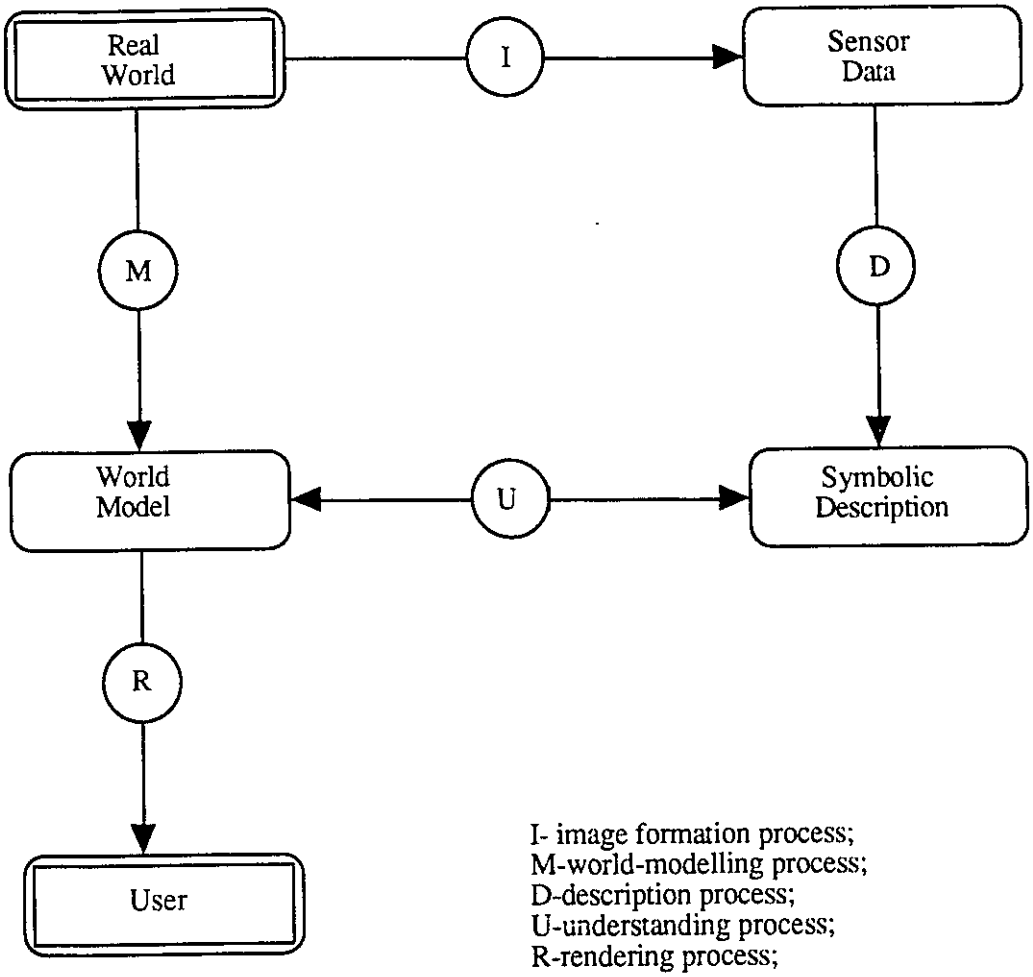


Figure 1.1. General object-recognition framework

about the real world. No a priori assumptions about real-world objects shapes should be incorporated

- *The modelling process* (M) creates object models for real-world objects. These geometrical objects models are a priori constructed by the system designer
- *The understanding, or recognition, process* (U) involves an algorithm to perform matching between model and data description
- *The rendering process* (R) produces a synthetic object description from object models allowing the system to communicate its understanding of a scene to any person querying the system.

Characteristics of an Ideal System

According to [7], the main capabilities that might be realized by an ideal object recognition system in the near future are:

- The system should be able to handle sensor data from arbitrary viewing directions. This requires a view-independent modelling scheme that is compatible with recognition processing requirements
- The system must handle arbitrarily complicated real-world objects
- The system must handle an arbitrary combination of a large number of objects with arbitrary orientations and locations
- The system must be able to handle a certain amount of noise in the sensor data without a significant degradation in system performance
- The system should be able to analyze scenes quickly and correctly
- It should not be difficult for the system to modify the world model data in order to handle new objects and new situations
- It is desirable for the system to be able to express its confidence in its own understanding of the sensor data.

The performance of object-recognition systems could be measured using the number of errors made by the system in performing the tasks on standardized sets of digitized sensor data that challenge the capabilities mentioned in the problem definition.

The typical errors that are made by such an object-recognition system are:

- *Miss error*: An object's presence is not detected
- *False alarm error*: The presence of an object is indicated even though there is no evidence of this in the sensor data
- *Location error*: An object occurrence is correctly identified, but the location of the object is wrong
- *Orientation error*: The object occurrence and position are determined correctly, but the orientation is wrong.

The information in the literature makes it practically impossible to compare existing object-recognition systems quantitatively because different researchers do not evaluate their systems in any consistent manner.

1.2. Review of 3-D Object-Recognition Methods

Vision involves the physical elements of illumination, geometry, reflectivity, and image formation, as well as the intelligent aspects of recognition and understanding. Most computer vision research has been done using grey-scale intensity images or range images as input sensor data. The digitized intensity images are arrays of numbers that indicate the brightness from which information about depth is difficult to infer.

One solution to 3-D object recognition is to use range sensors. These sensors produce range data that quantify the distances from the sensor focal plane to the object surface.

Because for these kinds of sensors the depth information depends only on the geometry and is independent of illumination and reflectivity, intensity-image problems with shadows and surface marking do not occur. Data from multiple range sensors can be merged into 2-D or 3-D range image or depth maps. One can expect the 3-D object recognition to be less difficult in range images than in intensity images.

The classical methods (*unstructured methods*) have tried to solve the problem of 3-D object recognition using only the information available in the grey-scale intensity image of the objects. *Image structuring methods* are based on using supplemental information in the scene to make the object recognition problem less difficult.

Unstructured methods

These methods use only the information contained in the images of physical objects. Much effort has been devoted in the last few years for solving the problem of 3-D object recognition from 2-D grey-scale intensity images. Techniques such as *shape from shading, shape from texture,* and *shape from contour* (sometimes also known as *shape from shape*) have been developed to make this task possible for a vision system [2]. The shape from shape method is considered to be the most important one.

Barrow and Tenenbaum [3] have shown that in cases of conflict between contour and shading information, humans use the contour information. Beiderman [4] claims that in experiments with humans, the recognition of a full-colour image of an object is not faster than the recognition of the object. Thus, we can conclude that the shape from contour is an essential element in monocular perception that cannot be ignored.

Even if limited to polyhedral objects, inferring the 3-D structure from the 2-D shape by analyzing the drawings of polyhedra, the problems of segmentation and surface orientation have proved not to be easy to be solved. Some techniques have also been proposed for

non-polyhedral objects in [3], [5], [6]. Most of these techniques examine a single surface in the scene, whereas human perception of one surface can be strongly influenced by the perception of the other surfaces comprising the entire object.

Ulupinar and Nevatia [2] proposed a new technique based on the analysis of symmetries in a scene. They assumed that clean, closed boundaries are given (or can be extracted from the real image) and they used certain symmetries to infer surface orientation. However they did not address the issue of separating object boundaries from surface markings or other perceptual groupings.

Some stereo techniques that simulate the eyes of a human have also been developed. They use multiple sensors (two cameras for example) to estimate stereo disparity and then recover depth. The main difficulty with these methods is solving the correspondence problem defined in [7].

Image structuring methods

In these methods, external structuring is introduced in the image formation process in such a way that the supplementary information that is obtained can be used to infer space relationships between the objects in the image. Two methods that investigate the problem of 3-D object recognition in a structured environment are known.

Structured light: In illuminating the scene, natural ambient light is replaced by an artificial light source, which can be of any structure or pattern, such as stripes or dots, that is convenient for the task. This kind of grid encoding represents a versatile technique for 3-D object feature recovery from 2-D images [8]-[14]. Ambiguities may occur when using structured light because the order in which the light stripes/dots appear in the 2-D image is not necessarily the same as in the initial projection grid [11]. The usual way to solve this problem is to establish an a priori correspondence between the detected light

stripes/dots and their position in the projection grid. Different methods have been proposed to implement such a correspondence: grid node identification by space encoding of projected rays [10], [12], colour indexing of the grid lines [11], grid line labelling by thickness [12], grid line identification by using the knowledge about the constraints existent in the physical world [13], or grid line identification by pseudo-random binary sequences (PRBSs) encoding [14]. Once the observed grid stripes/nodes are indexed, a multitude of image processing techniques are available to recover their 3-D Position Orientation and Scaling Estimation (POSE) parameters such as in [15] and [16].

Permanent object surface marking: This method has been mostly used for Automated Guided Vehicles (AGV) navigation by permanently marking the floor with continuous guide paths. More recently PRBS 1-D encoding has been studied for AGV position recovery [27].

This thesis presents an extension of these encoding techniques to 3-D object recognition. This new method is based on a permanent absolute-type encoding of the visible object surfaces with the terms (marked by distinct graphical symbols) of a pseudo-random array (PRA).

The problem of multiple object recognition using permanent pseudo-random array encoding can now be stated as follow:

Problem:

Given a set of 3-D objects having their visible surfaces marked with symbols representing terms of a PRA, *given* the mapping between the planar faces of the polyhedral models of these objects and the PRA, and *given* a monocular (2-D) image partly showing an object of this set, determine object identity and its POSE parameters relative to the 3-D frame of the video-camera.

Solution:

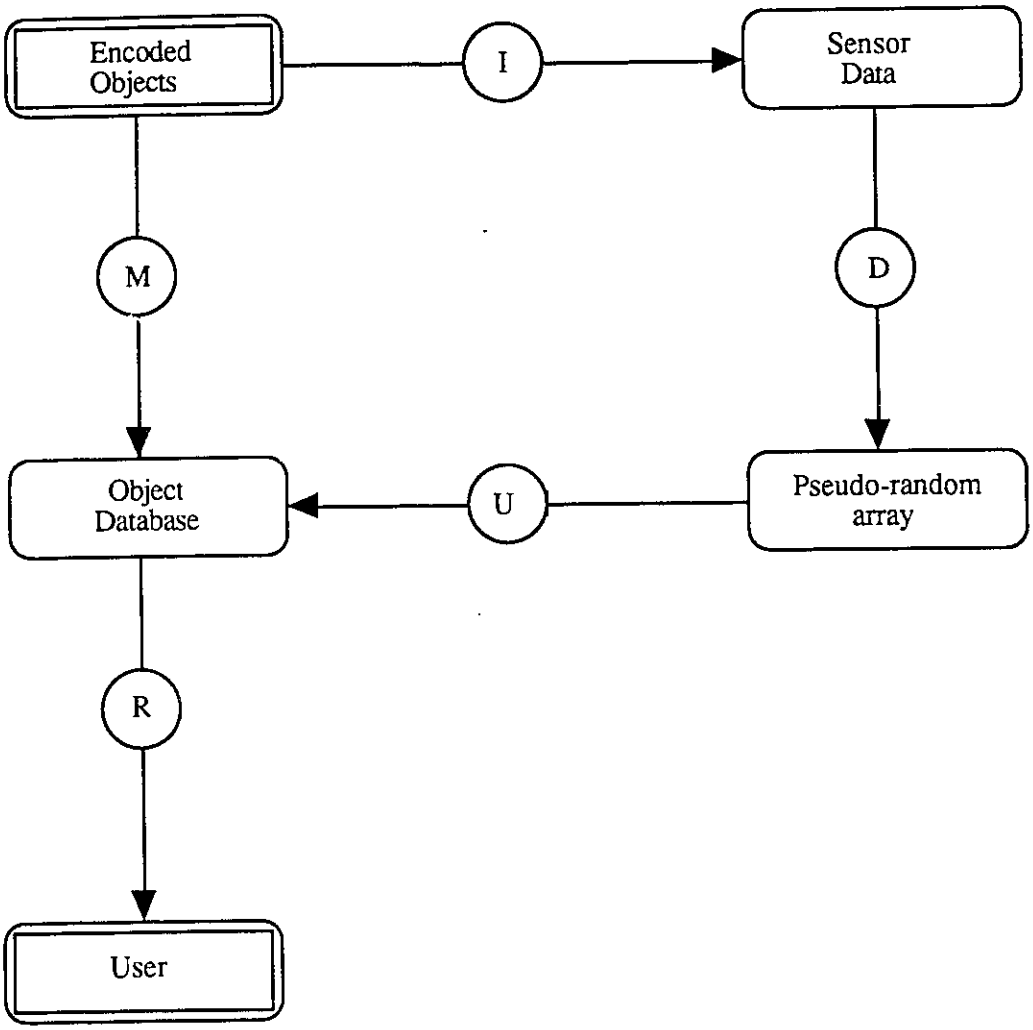
From the symbols which could be recognized in the image, a broken portion of the PRA is reconstructed and then inspected until a complete pseudo-random window is found. Such a window has the uniqueness property which allows the full identification of the window's absolute coordinates (i,j) within the PRA. Consulting the "object model/ PRA" mapping database it is possible to identify the object and its specific face containing the recovered window. Once the object face is identified, its POSE parameters are calculated from the perspective transformations relating the 2-D positions of the recognized symbols in the image frame to their correspondent PRA grid nodes defined in the 3-D object model frame (a classical "camera calibration" problem).

Figure 1.2 illustrates how this paradigm fits in the general object-recognition framework presented in Figure 1.1. Thus:

- The real world domain has been reduced to an artificial world domain of encoded objects. With this condition, the applicability of the method is restricted to applications such as space station, nuclear plants, or hazardous but controllable environments which allow for object encoding
- The sensor data domain consists of 2-D images of the encoded objects surfaces
- The modelling domain is implemented as a relational database (Object Database) and is presented in Chapter 4. It stores the information about the geometrical dimension of the object and also the mapping between the PRA and objects surfaces
- The intermediate symbolic scene description is represented by a portion of the PRA recovered from grey-scale intensity images.

The meanings of the processes that describe this structure are changed to:

- *The image-formation process* (I) creates grey-scale intensity images of PRA encoded objects surfaces
- *The description process* (D) is now implemented as a PRA symbol



I- image formation process;
M-world-modelling process-storing and retrieving objects from Object Database;
D-description process-PRA grid recognition;
U-understanding process-a query of the Object Database;
R-rendering process;

Figure 1.2. Object-recognition system structure for encoded PRA symbols.

recognition technique followed by a pseudo-random window index recovery and is presented in Chapter 3

- *The world modelling process* (M) is now the process of storing objects in the database
- *The understanding, or recognition process* (U) is in fact a query to the Object Database each time a pseudo-random window is recovered (this can be done based on the mathematical properties of PRA presented in Chapter 2)
- *The rendering process* has the same meaning as in the general object-recognition system presented in Figure 1.1.

PART 2

PSEUDO-RANDOM SEQUENCES AND ARRAYS

The approach to 3-D object visual recognition discussed in this thesis is based on a permanent encoding of the visible surfaces of all objects with symbols corresponding to the terms of a pseudo-random array. This chapter deals with pseudo-random sequences and arrays, how they are generated and some of their useful properties.

2.1. Pseudo-Random Sequences

Because the theory of pseudo-random sequences is based on the theory of finite fields a short review of finite fields is first given.

2.2.1. Finite (Galois) Fields

A field is a set of elements **F** equipped with two binary operations "+" (addition) and "x" (multiplication) such that for all $a, b, c \in \mathbf{F}$ the following axioms hold [17], [18], [22]:

a) **F** is an abelian group under addition ("+"):

1.) Addition in **F** is commutative

$$a+b=b+a;$$

2.) Addition in **F** is associative

$$a+(b+c)=(a+b)+c;$$

3.) There exists an element *0* belonging to *F* such that

$a+0=0+a=a$

for all *a* belonging to **F**;

4.) For each element *a* belonging to **F** there exists an element *-a* belonging to **F** such that

$a+(-a)=(-a)+a=0;$

b) $\mathbf{F}\backslash\{0\}$ is an abelian group under multiplication ("*x*"):

1.) Multiplication in **F** is commutative

$axb=bxa;$

2.) Multiplication in **F** is associative

$ax(bxc)=(axb)xc;$

3.) There exists an element *1* belonging to **F** such that

$ax1=1xa=a$

for all *a* belonging to **F**;

4.) For each element *a* belonging to **F** there exists an element a^{-1} belonging to **F** such that

$ax(a^{-1})=(a^{-1})xa=1;$

c) The following distributive laws hold:

1.) $(a+b)xc=(axc)+(bxc);$

2.) $ax(b+c)=(axb)+(axc);$

The symbol *0* is the identity element of the additive group and the symbol *1* is the identity element of the multiplicative group.

The rational numbers **Q**, real numbers **R**, and complex numbers **C** are examples of fields. These fields are called infinite fields because each one contains an infinite number of elements. A field that contains a finite number of elements is called a *finite field.* The number of elements in the field is called the order of the field. A field with *q* elements is also called a Galois field and is denoted by GF(*q*).

Construction of finite fields

The simplest class of finite fields are those with a prime number of elements. This class of fields is important because any finite field contains a subfield belonging to this class.

A field belonging to this class can be constructed in the following manner :

> Let p be any prime number. Then the integers modulo p under addition and multiplication modulo p form a field GF(p) with p elements . The elements of this fields are {0, 1, 2, ..., p-1}.

For p=5 the operations of addition and multiplication are given by Table 2.1 and 2.2:

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

Table 2.1. Addition table for GF(5).

x	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

Table 2.2. Multiplication table for GF(5).

An important property of this class of fields is that the elements of the field, except the element *0* (the identity element for addition), form a cyclic group under multiplication. This means that any finite field with a prime number of elements contains at least one element called a generator from which all the others elements of the field can be obtained. For example, the element 2 for GF(5) is a generator element for the cyclic group {1,2,3,4} under multiplication modulo 5:

$$2^1=2;\quad 2^2=4;\quad 2^3=3;\quad 2^4=1;$$

The class of finite fields that have a number of elements equal to a power of a prime is the most general class of finite fields. This class of fields are known as polynomial fields, or Galois fields of order $q=p^m$, and are denoted by GF(p^m).

A polynomial field with $q=p^m$ elements, where p is a prime and m is any positive integer, can be constructed as follows:

- the elements of the field are all polynomials in x of degree less then or equal to m-1 with coefficients from GF(p)
- addition is done in the ordinary way, modulo p
- multiplication is done by multiplying modulo p and taking the remainder when divided by a polynomial h(x).

The polynomial h(x) is any irreducible polynomial with coefficients from GF(p) of degree m. Also h(x) has to be irreducible to ensure that the inverse $a(x)^{-1}$ exists. Recall that a irreducible polynomial is a polynomial with coefficients from GF(p) which is not the product of two polynomials of lower degree.

The field constructed in this way contains p^m elements and is denoted by GF(p^m). It can be proven that a finite field of order p^m exists for all primes p and positive integers m, and that these are the only finite fields that exist. It can be also proved that all finite fields of order p^m are isomorphic. (Two finite fields are said to be isomorphic if there is a one-to-one mapping from one field to another which preserves the addition and multiplication operations) [18].

Consider the following example: p=3, m=3, $q=p^m$=27, $h(x)=x^3+2x+1$ and construct GF(27). The elements of the field are:

0, 1, 2, x, x+1, x+2, 2x, 2x+1, 2x+2, x^2, x^2+1, x^2+2, x^2+x, x^2+x+1, x^2+x+2, x^2+2x, x^2+2x+1, x^2+2x+2, $2x^2$, $2x^2+1$, $2x^2+2$, $2x^2+x$, $2x^2+x+1$, $2x^2+x+2$, $2x^2+2x$, $2x^2+2x+1$, $2x^2+2x+2$.

The coefficients of these polynomials are obtained by taking all possible three-element combinations of the elements of GF(3).

An example of polynomial adition in this field is like this:

$$(x^2+x+1) + (x^2+2x) = 2x^2+1$$
$$(x+1) + (2x^2+2x+2) = 2x^2$$

and polynomial multiplication is like this:

$$(x^2+2x)(2x^2+1)=2x^4+x^3+x^2+2x= (x^3+2x+1)(2x+1) + (x+2)=h(x)(2x+1) + (x+2)$$

so that

$$(x^2+2x)(2x^2+1)= (x+2) \quad \text{when} \quad h(x)=x^3+2x+1.$$

The elements of any finite field can be expressed in four different ways:

- as a m-tuple of elements belonging to GF(p)
- as a polynomial of degree lower or equal to m with coefficients from GF(p)
- as a power of t (where t is a primitive element, a zero for h(x))
- as a logarithm.

The irreducible polynomial h(x) used to generate a field is called the generator polynomial. It has been proved that for any positive integer m and any prime p there exists a primitive polynomial of degree m with coefficients from GF(p) [17].

The non-zero elements of the field form a cyclic group of order p^m-1 with generator t. Any generator for this cyclic group is called a primitive element for GF(p^m). Any irreducible polynomial having a primitive element as a zero is called a primitive polynomial. Not all irreducible polynomials are primitive. All finite fields contain a primitive element. Usually, the minimal polynomial of a primitive element of GF(p^m) has degree m and is chosen to be the generator polynomial for GF(p^m). Tables of primitive polynomials for some m and p can be found in [18] and [19].

The correspondence between the first two representations (m-tuple or polynomial) and the last two representations (power of t or logarithm) can be found as previously shown by dividing the polynomial x^n by the primitive polynomial h(x) and taking the reminder (note that t is zero for h(x)).

Table 2.3 gives the elements of GF(27) expressed in all four possible ways. The element t^4, for example, can be written as:

$$x^4=(x^3+2x+1)x+x^2+2x$$

so the correspondence is given by:

t^4--->021

000	0	0	$-\infty$
001	1	1	0
010	x	t	1
100	x^2	t^2	2
012	x+2	t^3	3
120	x^2+2x	t^4	4
212	$2x^2+x+2$	t^5	5
111	x^2+x+1	t^6	6
122	x^2+2x+2	t^7	7
202	$2x^2+2$	t^8	8
011	x+1	t^9	9
110	x^2+x	t^{10}	10
112	x^2+x+2	t^{11}	11
102	x^2+2	t^{12}	12
002	2	t^{13}	13
020	2x	t^{14}	14
200	$2x^2$	t^{15}	15
021	2x+1	t^{16}	16
210	$2x^2+x$	t^{17}	17
121	x^2+2x+1	t^{18}	18
222	$2x^2+2x+2$	t^{19}	19
211	$2x^2+x+1$	t^{20}	20
101	x^2+1	t^{21}	21
022	2x+2	t^{22}	22
220	$2x^2+2x$	t^{23}	23
221	$2x^2+2x+1$	t^{24}	24
201	$2x^2+1$	t^{25}	25

Table 2.3. The elements of the GF(27) generated by $h(x)=x^3+2x+1$, where t is a root of h(x).

A element t^n can be obtained from the previous one t^{n-1} by multiplying it with t. If the result is a polynomial with a degree greater than m the same procedure is applied, that is the result is divided by h(x) and only the reminder is taken. For example

$$x^{18}=x^{17}x=(2x^2+x)x=2x^3+x^2=h(x)x+(x^2+2x+1)$$

More tables that give the elements of the following Galois fields: GF(2), GF(2^2), GF(2^3), GF(2^4), GF(2^5), GF(2^6), GF(3), GF(3^2) are given in the Appendix A2.

2.1.2. Pseudo-Random Sequences and Their Propertie

Feedback shift register and PRS generation

The correspondence between an element of the field expressed as a m-tuple and the same element expressed as a logarithm can be obtain using a modified feedback shift register as further explained (see Figure 2.3). The states of the feedback shift register represent an index list, a list of the elements of the field in m-tuple form. The number of steps needed to arrive in any given state represent the logarithmic expression of the Galois field element corresponding to the given state.

For any given primitive polynomial h(x) of degree m with coefficients from GF(p) a feedback shift register can be constructed as follows:

- the number of memory cells is equal to m
- each memory cell contains an element of the field GF(p)
- at each time unit the contents of the memory cells are shifted one place to the right
- the content of the left most memory element is the sum of the terms corresponding tó h(x) (all the operations are done in GF(p)).

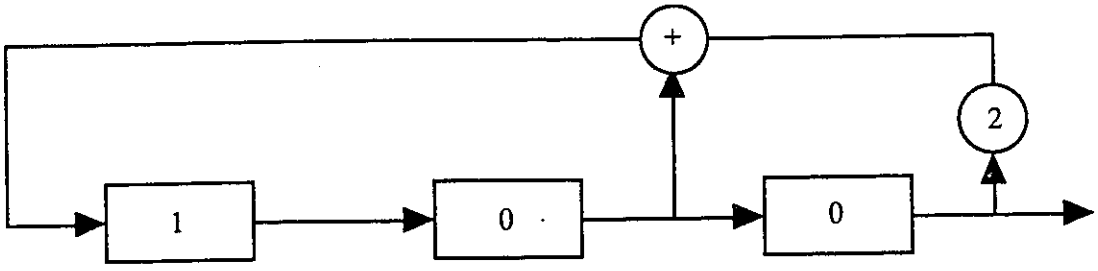


Figure 2.1. Feedback shift register for GF(27) with $h(x)=x^3+2x+1$.

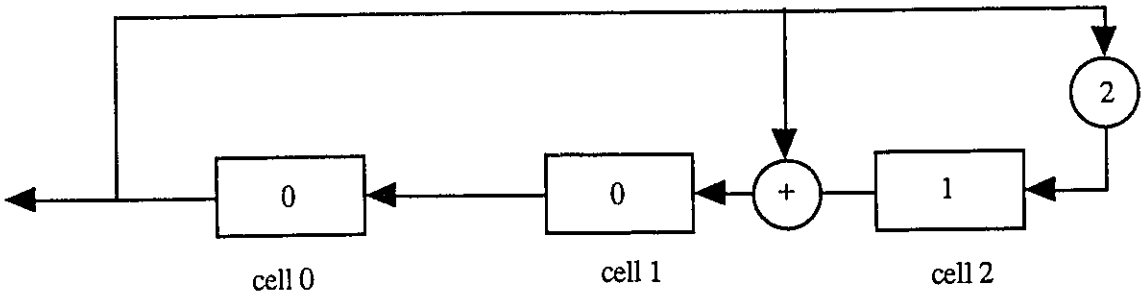


Figure 2.2. The modified feedback shift register for GF(27).

Figure 2.1 shows, the feedback shift register that gives GF(3^3) when $h(x)=x^3+2x+1$. The corresponding feedback equation is:

$$x_{i+3}=x_{i+1}+2x_i$$

and $x_2=1$, $x_1=0$, $x_0=0$ is the arbitrary initial state. All the operations are done in GF(3).

Pseudo-random sequences (PRS) which are also called pseudo-noise (PN) or m-sequences are sequentially generated as the output of a shift register. To obtain the correspondence between the elements of the Galois field (see Table 2.3), the modified feedback shift register shown in Figure 2.2 has to be used. It represents the reverse of the shift register presented in Figure 2.1. The states of this shift register provide a list of the elements of GF(3^3), both as successive powers of t and as m-tuple or polynomial in x of degree at most three, where only the coefficients of the polynomials are given. The next example presented in Figure 2.3 illustrates the generation of PRA for GF(27) with 001 as the initial m-tuple state. The states of the modified feedback shift register and the regular feedback shift register are also given. The contents of the cells of the modified feedback shift register are obtained according to the following equations, which result directly from Figure 2.2 :

$$(\text{cell } 0)_k=(\text{cell } 1)_{k-1}$$

$$(\text{cell } 1)_k=(\text{cell } 0)_{k-1}+(\text{cell } 2)_{k-1}$$

$$(\text{cell } 2)_k=2(\text{cell } 0)_{k-1}$$

As shown in Figure 2.3 all the elements of GF(3^3) appear in this development just once. Each line is identical with the previous one with the exception that the last symbol of the sequence is obtained as the output of the modified feedback shift register, and the state of the register is obtained according to the equation presented above. An index can be associated with each line and it represents the numbers of steps required for a shift register to arrive in this state starting from the original state 001. These indexes are in fact the logarithmic representation of the elements of the Galois field.

clock step	PRS	state of the modified feedback shift register	state of the shift register
0	0	**001**	**001**
1	00	**010**	**010**
2	001	**100**	**101**
3	0010	**012**	**112**
4	00101	**120**	**121**
5	001012	**212**	**211**
6	0010121	**111**	**112**
7	00101211	**122**	**120**
8	001012112	**202**	**201**
9	0010121120	**011**	**011**
10	00101211201	**110**	**111**
11	001012112011	**112**	**110**
12	0010121120111	**102**	**100**
13	00101211201110	**002**	**002**
14	001012112011100	**020**	**020**
15	0010121120111002	**200**	**202**
16	00101211201110020	**021**	**021**
17	001012112011100202	**210**	**212**
18	0010121120111002021	**121**	**122**
19	00101211201110020212	**222**	**221**
20	001012112011100202122	**211**	**210**
21	0010121120111002021221	**101**	**102**
22	00101211201110020212210	**022**	**022**
23	001012112011100202122102	**220**	**222**
24	0010121120111002021221022	**221**	**220**
25	00101211201110020212210222	**201**	**200**

Figure 2.3. The generation of a PRS over GF(3) of length 3^3-1.

For each finite field GF(p^m) there are p^m-1 PRSs that are basically the same sequence shifted to the right or to the left by a number of symbols depending on the state of the register when the sequence generation was started.

For all our next discussion, we consider for each field only the sequence that starts with the element corresponding to the primitive element of the field. This sequence has the property that ensures the correspondence between the two forms of expressing the elements of the Galois field (the polynomial form and the logarithmic form). By using this sequence, pseudo-random to natural decoding is equivalent to finding the corresponding logarithm representation for a given element known in its polynomial representation.

Since there are m memory elements containing numbers between 0 and p-1, there are p^m possible states for the shift register. Thus, the infinite sequence that the shift register will generate is periodic. The state in which all the memory elements are zero is not included because the shift register cannot leave this state. Hence, the maximum length of the sequence is p^m-1. This is the reason why the pseudo-random sequences are also called maximum-length shift register sequences.

Properties of PRS

Let h(x) be a fixed primitive polynomial of degree m with coefficients from GF(p), and let $\delta_m(q)$ be the set consisting of the pseudo-random sequences obtained from h(x). We proceed to give the property of these sequences [19]:

Property 1. The Shift Property: If $b=b_0\ b_1\ \ldots\ b_{q-2}$ is any PRS in $\delta_m(q)$, then any cyclic shift of b, say

$$b_j\ b_{j+1},\ \ldots\ b_{q-2}\ b_0\ \ldots\ b_{j-1}$$

is also in $\delta_m(q)$.

Property 2. The Recurrence Property: Any PRS $b \in \delta_m(q)$ satisfies the recurrence

$$b_{i+m}=h_{m-1}b_{i+m-1}+h_{m-2}b_{i+m-2}+\ldots+h_1b_{i+1}+b_i \qquad i=0,\ 1,\ \ldots$$

Property 3. The Window Property: If a window of width m is slide along a PRS in $\delta_m(q)$, each of the p^m-1 non-zero binary m-tuple is seen exactly once. (This follows exactly from the fact that h(x) is a primitive polynomial.) Let us consider the PRS generated by the primitive polynomial $h(x)=x^3+2x+1$ over GF(3). As previously shown a cycle of this sequence is given by:

0 0 1 0 1 2 1 1 **2 0 1** 1 1 0 0 2 0 2 1 2 2 1 0 2 2 2

and the three bolded symbols form a window that is unique in this sequence.

Note: To avoid difficulties at the ends of the sequences either imagine that three copies of the sequence are placed next to each other, or that the PRS is written in a circle.

Property 4. The Pseudo-Random Property: In any PRS in $\delta_m(q)$ 0 occurs q^{m-1} -1 times and every non-zero element of GF(q) occurs q^{m-1} .

Property 5. The Addition Property: The sum of two sequences in $\delta_m(q)$ (formed component-wise, modulo m, without carries) is another sequence in $\delta_m(q)$. For instance, the sum of the first two sequences given above in the set of 26 sequences is the tenth sequence.

Property 6. The Shift-and-Add Property: The sum of a pseudo-random sequence and a cyclic shift of itself is another PRS. (From the shift property and the addition property.)

Property 7a: A PRS in $\delta_m(q)$ has the form

$$a=b, \gamma b, \gamma^2 b, \ldots, \gamma^{q-2} b,$$

where b is a sequence of length $(q^m-1)/(q-1)$ and γ is a primitive element of GF(q). This is because the states of the shift register can be made to correspond to a logarithmic table

of GF(q).

Property 7b: Let $a=(a_0 \ldots a_{n-1})$ be a PRS in $\delta_m(q)$, and let $b=(a_s\ a_{s+1} \ldots a_{s-1})=(b_0 \ldots b_{n-1})$ be a shift of a by s places. If s is not a multiple of q-1, then among the q^m-1 pairs (a_i,b_i) , (0, 0) occurs q^{m-2}-1 times and every other pair of elements of GF(q) occurs q^{m-2} times.

Property 7c. Autocorrelation Function Property: The autocorrelation function of a complex-valued sequence â obtained from $a \in \delta_m(q)$ by replacing $r \in$ GF(q) by $e^{2\pi ir/q}$ is given by

$$\rho(0)=1,\ \rho(i)= -(1/(q^m-1)\ ,\quad 1 \le i \le q^m-2.$$

For other autocorrelation functions, all of them can be calculated from Property 7b [23], [24].

Note: The normalized autocorrelation function $\rho(i)$ of a real or complex sequence $s_0\ s_1,\ldots s_{n-1}$ of length n is defined by

$$\rho(i) = \frac{1}{n}\sum_{j=0}^{n-1} s_j \bar{s}_{i+j}$$

where subscripts are reduced modulo n if they exceed n-1, the bar denotes complex conjugation, and i=... -2, -1, 0, 1, 2, ...

For pseudo-random binary sequences PRBS (sequences defined over GF(2) the autocorrelation function is given by:

$$\rho(0) = 1$$
$$\rho(i) = -1/n$$

and it can be shown [25] that this is the best possible autocorrelation function of any binary sequence of length 2^m-1, in the sense of minimizing the maximum value of $\rho(i)$.

2.1.3. 1-D Pseudo-Random Window Index Recovery

Let h(x) be a polynomial of degree *m* with coefficients from GF(p), and $a \in \delta_m(q=p^m)$, a PRS that has the property that it is generated by a modified feedback shift register starting from an initial state is identical with the element corresponding to the primitive element of the field. Let us also consider a m-size window belonging to this sequence, represented using bold characters below, where the element $\mathbf{a_k}$ is considered to be the origin of the window.

$$a=a_0\ a_1\ a_2 \cdots \mathbf{a_k}\ \mathbf{a_{k+1}}\ \mathbf{a_{k+2}} \cdots \mathbf{a_{k+m-1}}\ a_{k+m} \cdots a_{q-2}$$

The problem of window index recovery can be stated as:

> *Given the elements of an m-size window belonging to a PRS generated by h(x) over GF($q=p^m$) determine the k index of the origin of the window* a_k.

Due to the fact that a window represents an element of GF(q) in its m-tuple form, the problem can be stated as determining the logarithm representation of a given element of the field when the m-tuple form is known.

This translation is always necessary for practical applications that use PRSs for encoding. Also, as we will see later the last step for 2-D pseudo-random window index recovery is reducible to a 1-D pseudo-random window index recovery.

Few methods are known in the literature that can be applied for solving this problem:

- A serial-type code conversion algorithm, extensively discussed in [26] exploits the reversibility of the PRS generating algorithm. This method is based on the idea that it is possible to find the index (the logarithm representation) associated with any pseudo-random m-tuple by simply counting the number

of reverse feedback shifts that it takes for the given m-tuple to shift back into the initial state of the shift register.

- A strictly parallel solution would be to use a code conversion table stored in ROM. This is expensive for applications requiring high encoding resolutions.

- A compromise solution [27] is a combination of the serial and parallel methods as exemplified in Figure 2.4. Consider a pseudo-random encoded track where certain positions (uniformly distributed with a period of t) are employed as "milestones". The code conversion for any position p=m*t+r, where m*t is the position of the nearest "down the track milestone" Q(m) and r represents the position relative to this milestone, will be discussed. The natural code for r is found by counting the steps required to arrive by successive back-shifts from the initial code to the nearest milestone Q(m). All intermediate states of this serial shift-back operation are checked in parallel against all possible milestone pseudo random patterns. Thus with this method the code conversion of the relative position r distance is found serially while the milestone code conversion is done in parallel.

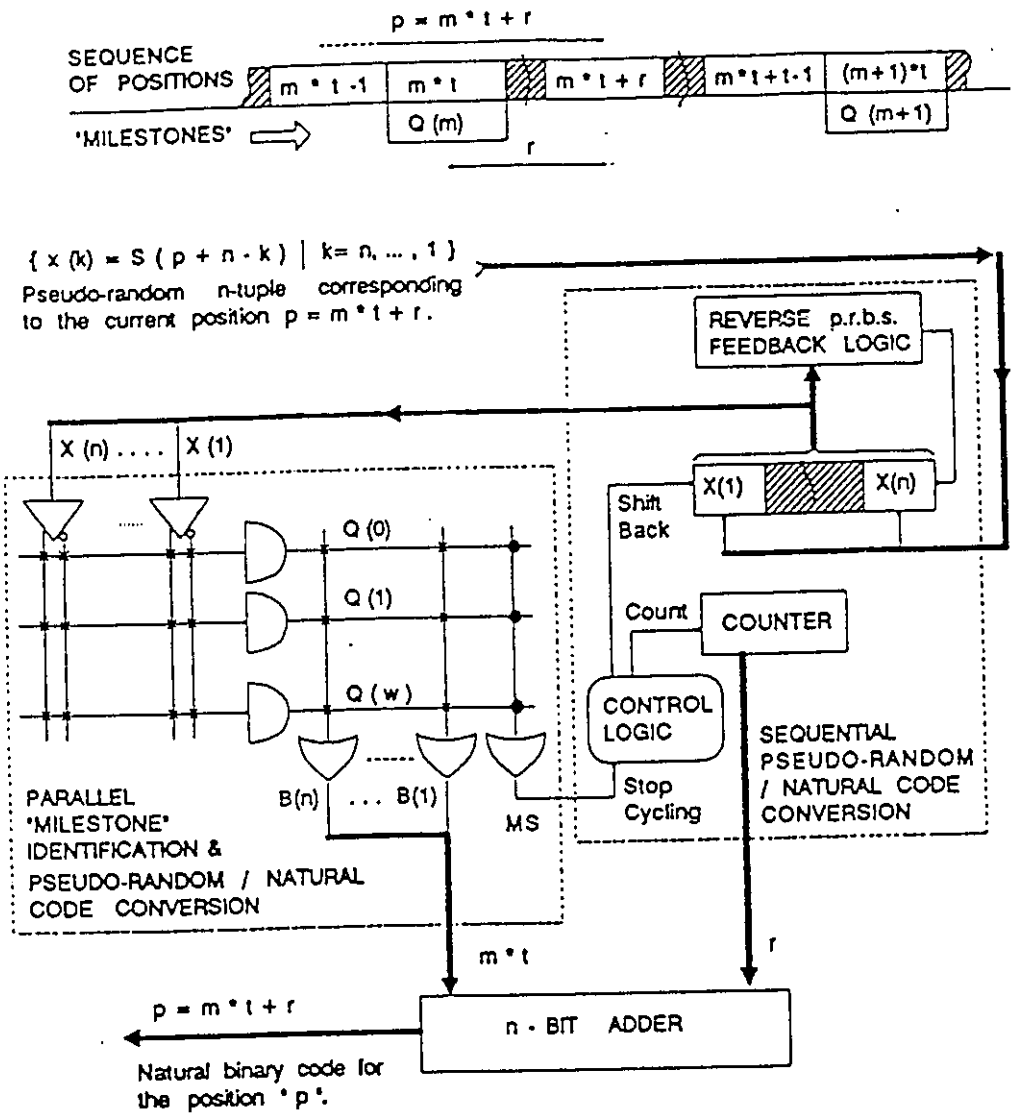


Figure 2.4. Serial-parallel pseudo-random/natural code conversion.

2.2. Pseudo-Random Arrays

A pseudo-random array (PRA) can be constructed by folding a PRS, (see the first three methods presented next), or by starting from two PRS and using a cross product to obtain the values of the elements of the PRA (see method 4).

2.2.1. Pseudo-Random Arrays and Their Properties

Let us start from a PRS defined by the primitive polynomial h(x) of degree m over GF(p), of length $l=p^m-1$. Three methods to construct a PRA starting from this PRS are presented next.

Method 1.

If n_1 and n_2 are two natural numbers so that $n_1 n_2=l$, then it is possible to construct a PRA of dimension n_1 by n_2 by taking the first n_1 elements of the PRS and filling the first row of the array, taking the next n_1 elements and filling the second row, and so on until the n_2 row is filled.

A window in this array is defined exactly as before as being any m-tuple belonging to a row. This window is unique in the array because it is unique in the PRS. This method is not very efficient because the array does not behave the same for both directions x and y. As well, a strip of width m-1 is lost on the right side of the array (an m-1 tuple is not unique in the array).

Note: The same procedure can be followed to fill the columns of the array. The first n_2 elements of the array are taken and used to fill the first column, the next n_2 for the second columns and so on. The last column, the n_1-th one is filled with the last n_2 elements of the PRS.

The PRS defined over GF(3) by the primitive polynomial $h(x)=x^4+x+2$ of length 80

00010021011120022010221101012122120122220002001202221001102011220202121121021111

can be used to construct an 8-by-10 PRA using this method as shown in Figure 2.5:

00010021
01112002
20102211
01012122
12012222
00020012
02221001
10201122
02021211
21021111

Figure 2.5. An 8-by-10 PRA constructed using method 1.

Method 2.

This method was suggested by Spann [28] and can be applied when the whole plane is to be filled with copies of the array. Using copies of the array constructed as described in the first method, the whole plane can be filled. The *m*-1 width strip is not lost anymore because the right neighbours of the array help to form an m-tuple (m-size window). Figure 2.6 shows how a PRA can be constructed using the PRS generated by $h(x)=x^4+x+1$, over GF(2), and Figure 2.7 shows the general case of constructing PRA using this method.

000100110101111000100110101111000100110101111
100110101111000100110101111000100110101111000
110101111000100110101111000100110101111000100
101111000100110101111000100110101111000100110
111000100110101111000100110101111000100110101
000100110101111000100110101111000100110101111
100110101111000100110101111000100110101111000
110101111000100110101111000100110101111000100
101111000100110101111000100110101111000100110

Figure 2.6. A PRA constructed using method 2.

a3	a4 a5 a6 a7	a8 a9 a10 a11	a12a13a14 a15	a0 a1 a2 a3	a4 a5 a6 a7	a8
a7	a8 a9 a10 a11	a12a13a14a15	a0 a1 a2 a3	a4 a5 a6 a7	a8 a9 a10 a11	a12
a11	a12a13a14 a15	a0 a1 a2 a3	a4 a5 a6 a7	a8 a9 a10 a11	a12 a13 a14 a15	a0
a15	a0 a1 a2 a3	a4 a5 a6 a7	a8 a9 a10 a11	a12 a13 a14 a15	a0 a1 a2 a3	a4
a3	a4 a5 a6 a7	a8 a9 a10 a11	a12a13a14 a15	a0 a1 a2 a3	a4 a5 a6 a7	a8
a7	a8 a9 a10 a11	a12a13a14a15	a0 a1 a2 a3	a4 a5 a6 a7	a8 a9 a10 a11	a12
a11	a12a13a14 a15	a0 a1 a2 a3	a4 a5 a6 a7	a8 a9 a10 a11	a12 a13 a14 a15	a0

Figure 2.7. The PRA construction using method 2.

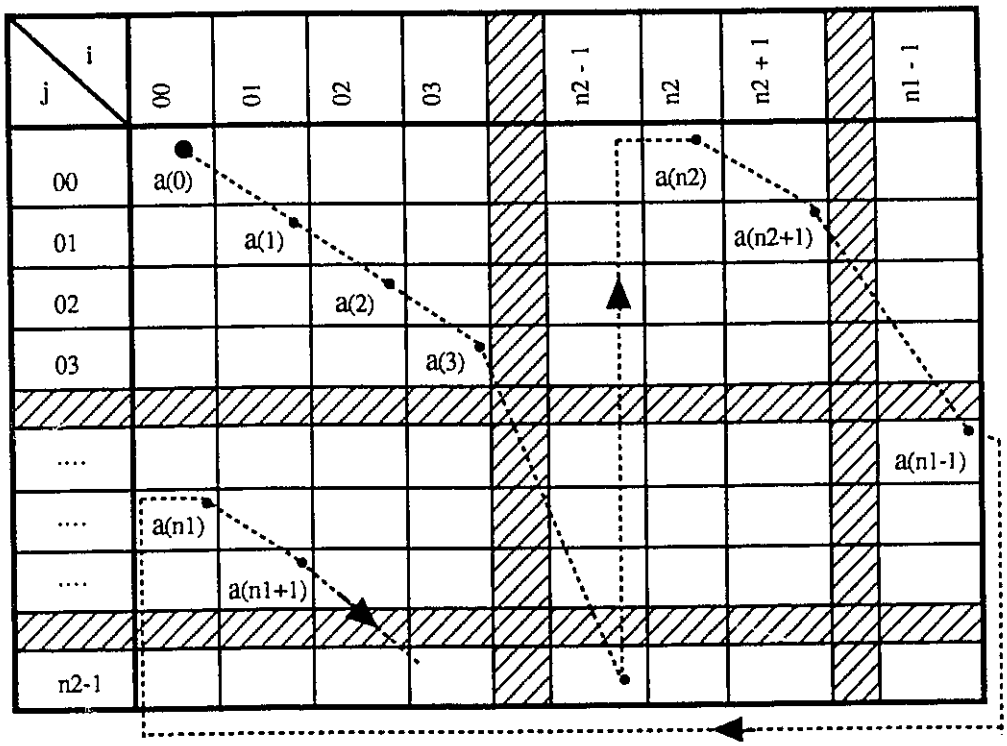


Figure 2.8. The PRA construction using method 3.

Method 3.

Using an idea from the first method, another method was developed but the window has two independent dimensions.

If $l=p^m-1$, then two numbers k_1 and k_2 are chosen such that $k_1k_2=m$, and the dimensions of the PRA are given by

$$n_1=p^{k1}-1 \text{ and } n_2=l/n_1 .$$

Note: $p^{k1}-1$ always divides $p^{k1k2}-1$.

Now the array is filled by writing the sequence down the main diagonal and continuing from the opposite side whenever an edge is reached (see [19]). Figure 2.8 presents this method of constructing PRAs and Figure 2.9 shows a 17-by 15 PRA obtained from a PRA defined by $h(x)=x^4+x^2+\omega\, x+\omega^2$ over GF(4) where ω is the generating element for $GF(2^2)$.

02132333113332312
00133210330123310
03002231331322003
01212032002302121
03320301111030233
03213111221113123
00211320110231120
01003312112133001
02323013003103232
01130102222010311
01321222332221231
00322130220312230
02001123223211002
03131021001201313
02210203333020122

Figure 2.9. A 17-by-15 PRA with entries from the field GF(4).

Although the PRS appears random to the eye, the corresponding arrays constructed in this manner have a conspicuous non-random feature, being symmetric about a column of zeros. They are called PRA because they share the same properties of the PRS (Properties 3, 4, and 7).

For real applications the PRAs constructed using these method have a big disadvantage: a window can be recovered if and only if a block of m symbols are recognized. One missed symbol in the centre of a portion of an array can make the recovery process impossible even if all the others are recognized.

Due to the ratio factor between the x and y dimensions of the pseudo-random window, these methods are not suitable for graphical encoding of the elements of the PRA. From a visual sensor (camera) a square image can be obtained and a square window is preferable.

Method 4.

A new method of constructing PRA has been recently proposed [37]. As we will see later this method has some advantages over the others that are very useful for our application.

Let us suppose that a and b are two PRSs defined by two primitive polynomials of degree m and n, over GF(p_1) and respective GF(p_2).

PRS1: $a_0, a_1, a_2, ..., a_u$

PRS2: $b_0, b_1, b_2, ..., b_v$

Let us consider A= {A1, A2, ..., Ap_1 } and B={B1, B2, ..., Bp_2 } to be the sets of elements of GF(p_1) and GF(p_2). The following relations hold:

$u=p_1^m-1$

$v=p_2^n-1$

where:

u-is the length of the PRS1 generated over GF(p_1);

v-is the length of the PRS2 generated over GF(p_2);

If p_1 is the number of symbols used to generate PRS1 and p_2 is the number of symbols used to generate PRS2 we have to use a set C={ C1, C2, ..., Cp} of $p=p_1p_2$ symbols to generate the PRA. These symbols are generated using the sets A and B (see Table 2.4).

	A1	A2	...	Ap_1
B1	C1	C2	...	Cp_1
B2	Cp_1+1	Cp_1+2	...	$C2p_1$
B3	$C2p_1+1$	$C2p_1+2$	...	$C3p_1$
...	...	...	...	...
Bp_2	$C(p_2-1)p_1+1$	$C(p_2-1)p_1+2$	...	Cp_1p_2

Table 2.4. The cross-product table for PRA generation using method 4.

Using these two PRSs and the Table 2.4 a PRA can be constructed as shown in Table 2.5. The elements of the arrays are $c_{i,j}$ and they belong to the set C.

	a_0	a_1	a_2	...	a_{n-1}
b_0	$c_{0,0}$	$c_{0,1}$	$c_{0,2}$	...	$c_{0,n-1}$
b_1	$c_{1,0}$	$c_{1,1}$	$c_{1,2}$	...	$c_{1,n-1}$
b_2	$c_{2,0}$	$c_{2,1}$	$c_{2,2}$	...	$c_{2,n-1}$
...	...	...	...	...	...
b_{m-1}	$c_{m-1,0}$	$c_{m-1,1}$	$c_{m-1,2}$	...	$c_{m-1,m-1}$

Table 2.5. The PRA construction using method 4.

Any rectangle of dimensions m and n in this array, which contains a valid symbol on each line and each column, will constitute a window. We defined the top, left symbol of the

rectangle W(i,j) to be *the origin of the window* (i, and j are the coordinates of this symbol relative to the top left corner of the PRA) even if this symbol is not a valid symbol for a particular case. The main drawback of this method is the fact that it produces a poorer resolution than method 3. However, this is counter balanced by the superior equipment/time cost of the pseudo-random/natural code conversion in the case of PRA constructed using this method. Requiring recognition of only one symbol in each row and column of the window, this encoding method is also more robust than method 3, which requires recognition of all the symbols in the window.

Also, the PRS1 and PRS2 can be the same sequence, or similar length sequences, and for these cases a square or close to square PRA-s and respective windows are obtained. For image processing, where rectangular images are usually processed, this will ensure that a maximum amount of information will be used. Another disadvantage of this method is that the recognition procedure has to be more complex since the number of symbols has increased.

Figure 2.10 shows an experimental PRA that was constructed using this method. The PRBS1 is defined by $h_1(x)=x^6+x+1$ over GF(2) and has length $u=2^6-1=63$, and PRBS2 is defined by $h_2(x)=x^2+x+1$ over GF(2) and has the length $v=2^4-1=15$. A number of different window contents (in bold) which provide enough information to recover the i and j coordinates for the random window W(24, 7) of size 6 by 4 are also given.

2.2.2. 2-D Pseudo-Random Window Index Recovery

The problem of 2-D window index recovery can be stated as :

> *Given enough elements to reconstruct a pseudo-random window that belong to a given PRA, determine the i and j indexes of the origin of the window W(i, j) relative to the top left corner of the PRA.*

000001000011000101001111010001110010010110111011001101010111111

0 000001000011000101001111010001110010010110111011001101010111111
0 000001000011000101001111010001110010010110111011001101010111111
0 000001000011000101001111010001110010010110111011001101010111111
1 222223222233222323223333232223333223223233233323332232333333333
0 000001000011000101001111010001110010010110111011001101010111111
0 000001000011000101001111010001110010010110111011001101010111111
1 222223222233222323223333232223333223223233233323332232333333333
1 222223222233222323223333232223333223223233233323332232333333333
0 000001000011000101001111010001110010010110111011001101010111111
1 222223222233222323223333232222333223223233233323332232333333333
0 000001000011000101001111010001110010010110111011001101010111111
1 222223222233222323223333232222333223223233233323332232333333333
1 222223222233222323223333232222333223223233233323332232333333333
1 222223222233222323223333232222333223223233233323332232333333333

W(24,7)-an example of a pseudo-random window;

3 2 3 2 2 2	3 2 3 2 2 2	3 2 3 2 2 2	3 2 3 2 2 2	3 2 3 2 2 2
1 0 1 0 0 0	1 0 1 0 0 0	1 0 1 0 0 0	1 0 1 0 0 0	1 0 1 0 0 0
3 2 3 2 2 2	3 2 3 2 2 2	3 2 3 2 2 2	3 2 3 2 2 2	3 2 3 2 2 2
3 2 3 2 2 2	3 2 3 2 2 2	3 2 3 2 2 2	3 2 3 2 2 2	3 2 3 2 2 2

Figure 2.10. An experimental PRA and some cases of a pseudo-random window recovery

The i and j indexes of a window for the PRA that were generated using method 1 can be obtained by recovering the PRW index in the corresponding PRS and by dividing this index with n_1 . Next, the i index is given by the reminder and the j index by the quotient of this division. Figure 2.5 shows the PRW(3,4) corresponding to a PRS index equal with 35.

Next we present the solution to 2-D index recovery for PRA generated using methods 3 and 4.

Let **a** be a PRS defined by the primitive polynomial h(x) of degree m over GF(p) of length $l=p^m-1$ given by:

$$\mathbf{a}=a_0\ a_1\ a_2\ a_3\ \ldots\ a_i\ \ldots a_{l-1}$$

and **b** the PRA that has been constructed using method 3 described above:

$$b=\begin{bmatrix} b_{00} & b_{01} & \cdots & b_{0,n2-1} \\ b_{10} & b_{11} & \cdots & b_{1,n2-1} \\ \cdots & \cdots & \cdots & \cdots \\ b_{n1-1,0} & b_{n1-1,1} & \cdots & b_{n1-1,n2-1} \end{bmatrix}$$

Because of the way the PRA was constructed (see Figure 2.8.) the following relations hold:

$a_0=b_{00}$

$a_1=b_{11}$

$a_2=b_{22}$

...

$a_i=b_{i1i2}$

...

where

$$i=i_1 \pmod{n_1}, \qquad 0 \le i_1 < n_1$$
$$i=i_2 \pmod{n_2}, \qquad 0 \le i_2 < n_2.$$

Conversely, given i_1 and i_2 with $0 \le i_1 < n_1$ and $0 \le i_2 < n_2$ there is a unique value of i in the range $0 \le i < n=n_1n_2$ such that the above relations hold (see [17], [19]), since n_1 and n_2 are relatively prime. (This can be proved by using the Chinese Remainder Theorem). Thus, the origin of a 2-D window belonging to a PRA has a unique corespondent in the PRS that was used to generate the PRA.

The pseudo-random to natural code conversion for this type of PRA is implemented as a memory store table. The example presented in Figure 2.9 shows the PRW(5,4).

The 2-D index recovery for a window that belongs to a PRA constructed using method 4. can be done in parallel by doing a recovery for the line index and for the column index using one of the methods presented in section 2.1.3. An inverse transformation has to be done first from the PRA's symbols to the PRS's symbols using Table 2.7.

PART 3

ENCODING SYMBOLS AND THEIR IMAGE RECOGNITION

This chapter discusses an application of the PRA encoding for visual 3-D object recognition. First a set of graphical symbols is chosen which will be used to mark the elements of the PRA on the object surfaces. After that we present an image processing technique that can be used to recognize these symbols. Finally, we show how portions of the physical PRA can be reconstructed and inspected to identify a pseudo-random window.

3.1. Symbols Used for Permanent Object Surface Encoding

The previous chapter has presented the generation of PRAs, their properties and also how the i and j coordinates of a window can be recovered by knowing the content of the window. Now a set of graphical symbols has to be defined to mark the PRA numerical elements of the array on the objects surfaces. The geometrical shapes of these symbols have to ensure an easy and robust way of their recognition using image processing techniques.

3.1.1. Symbol Design: The Graphical Symbol Set.

The symbols have to meet the following requirements [37]:

1.) The shape of the symbol must have enough information to provide:

- the symbol type

- the symbol origin-a unique point in the symbol is further considered as the 2-D position of that symbol in the image
- the 2-D symbol orientation - two axes allow to estimate the directions of the four neighbours of the symbol (north, south, east, and west) and the symbol asymmetry
- the relative symbol dimension-that information used in correlation with the symbol orientation make possible to estimate the position of the four neighbours.

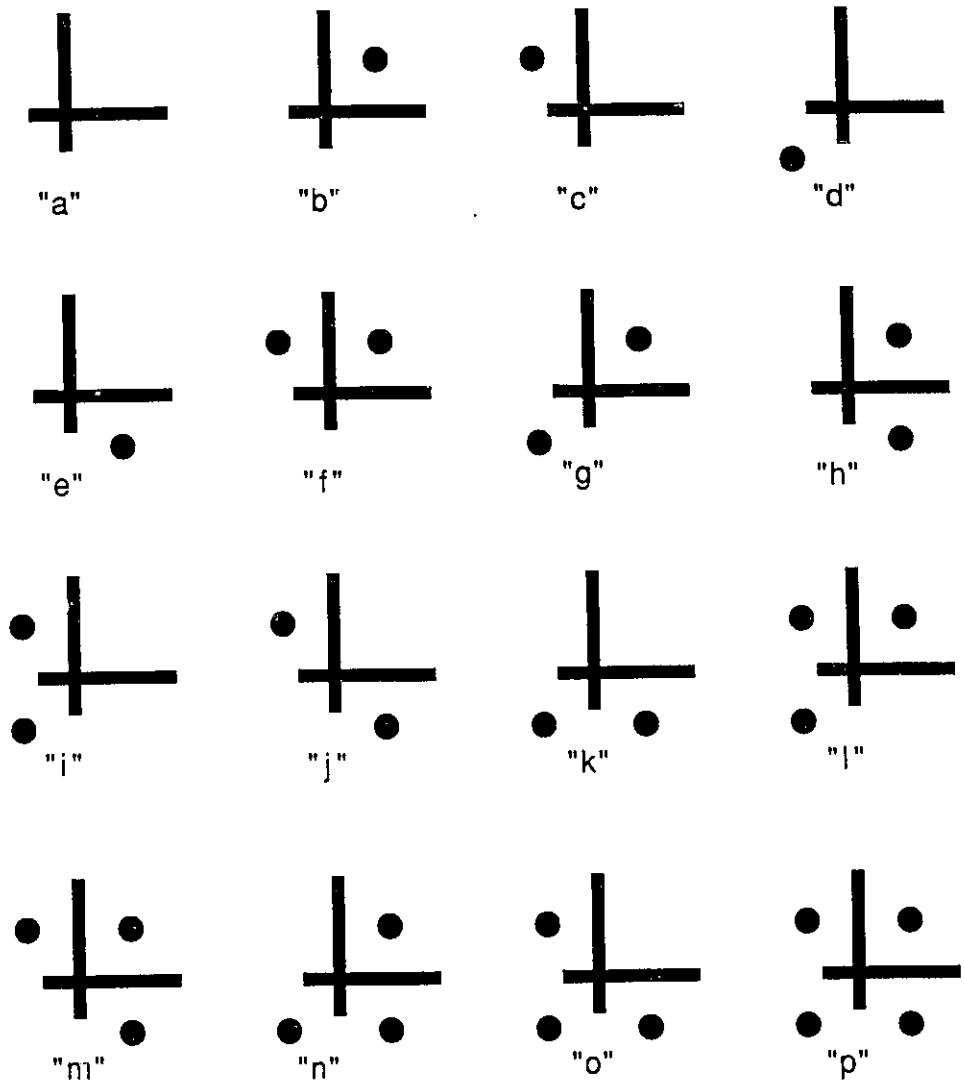
2.) Allow to implement a symbol recognition procedure that is invariant to position, orientation, scaling and perspective transformation of the symbols in a quick image processing technique.

3.) The symbols should have sufficient uniqueness so that other objects in the scene will not be mistaken for encoding symbols.

4.) There should be a *Hamming distance* (as defined on page 44) between each two symbols such that a symbol will not be recognized as being another due to the image processing noise.

The set of 16 graphical symbols S={*a, b, c, d, e, f, g, h, i, ,j, k, l, m, n, o, p*} which are shown in Figure 3.1 satisfies all of these requirements. The shape of these symbols has been designed to suit their image recognition by a skeleton method. Few other symbol types were tried [32], [33], but this symbol set has proved to be more efficient.

The geometrical dimensions and shape characteristics of a generic symbol of the set is presented in Figure 3.2. These characteristics are important for further symbol recognition and POSE recovery procedures.



S {a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p }

Figure 3.1. The 16 symobol set.

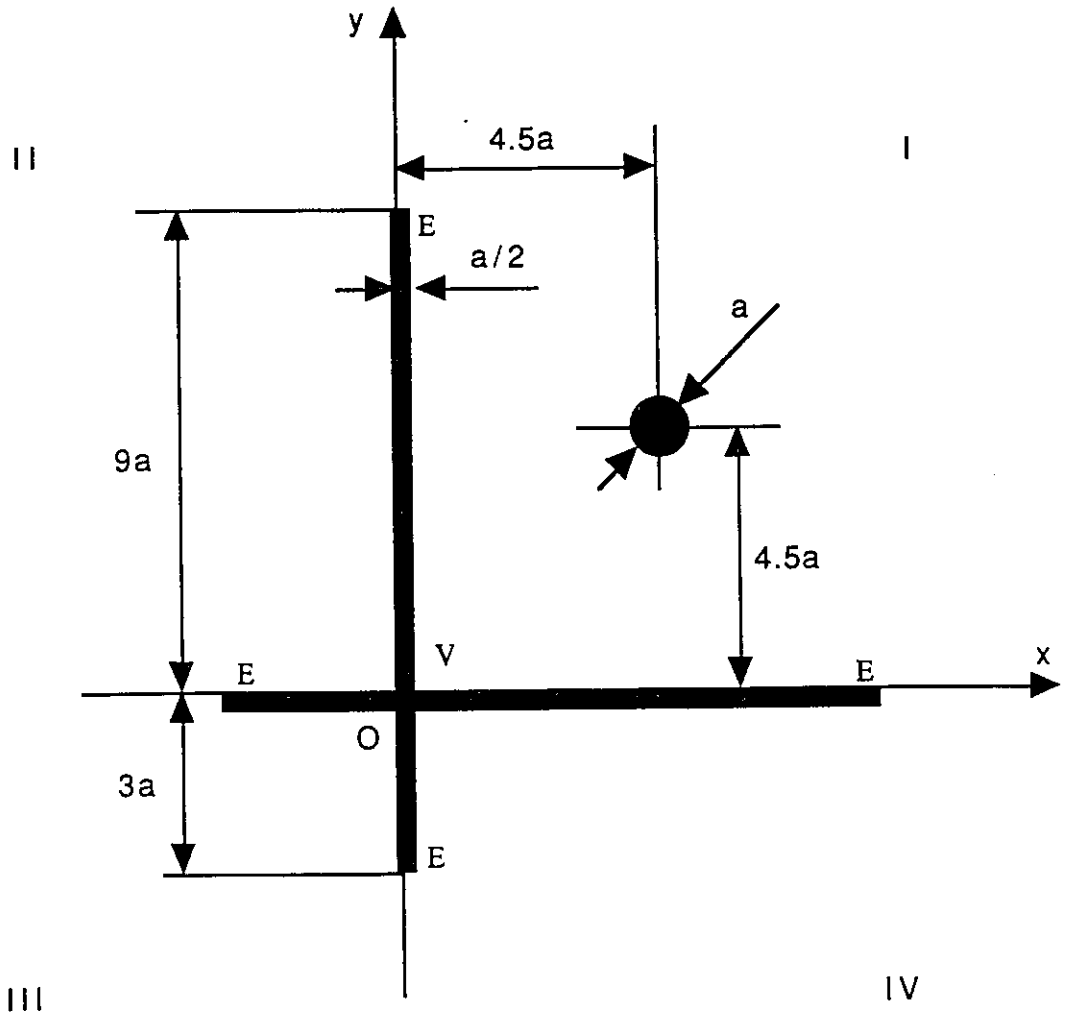


Figure 3.2. Symbol geometrical dimensions.

Subsets of these symbols can be selected to mark various PRAs. In order to have better noise immunity, the symbols have been selected in such a way as to have a *Hamming distance* between any two symbols of the subset of at least 2.

The *Hamming distance* D_{ij} between two symbols i and j of the set S is defined as being equal to the number of quadrants in which these two symbols differ. For example, we have:

$D_{ap}=4$; $D_{fg}=2$; $D_{bo}=4$; $D_{cd}=2$;

Figure 3.3 shows two subsets of S with four and six symbols respectively having a Hamming distance of two between each two symbols of each subset. These subsets will be further used to prove that such an approach to object recognition gives good results (see Chapter 6).

3.1.2. The Construction of the Physical PRA

Using a graphical symbol subset of the set S a "physical" PRA can be constructed. This geometrical PRA can be imagined to be like a "wallpaper" with the graphical symbols arranged in the vertices of a grid that encode each element of the numerical PRA.

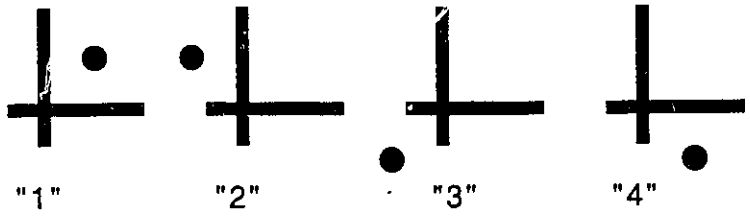
Each element of the physical array will have two pairs of coordinates:

- the (i, j) 2-D index values in the numerical PRA
- real positive coordinates (x, y) of the graphical symbol in the wallpaper.

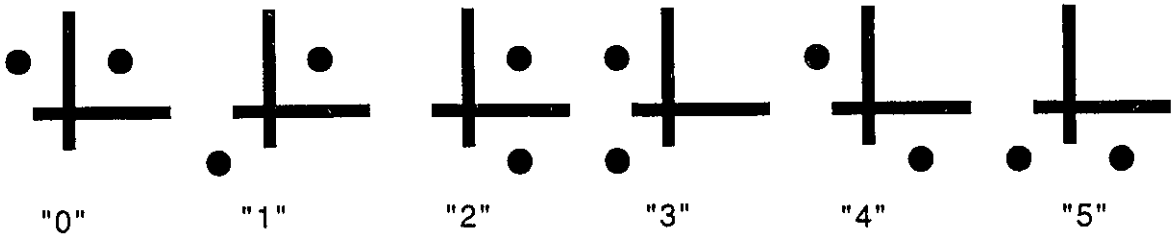
The following relations between the grid step **s**, the symbol dimension **d**, and the unit of the symbol **a** (see Figure 3.2.) have been chosen, by physical means.

d=12a

s=18a



$S_1\{0, 1, 2, 3\}$



$S_2\{0, 1, 2, 3, 4, 5\}$

Figure 3.3. Two subsets with Dij=2.

These empirical relations led to robust symbol image recognition. The value of **a** is a measure of the *density of symbols* (the number of symbols contained within a given surface). It has to be chosen in such a way to ensure that any surface that we want to encode will contain enough symbols to permit a window to be recovered. The parameter **a** does not have to be a constant; its value is proportional to the length of the smallest edge of a given face. As we will see later (Chapter 4), it will be stored in the Object Database and used for POSE recovery.

Figure 3.4 shows a portion of a physical PRA and the geometrical dimensions that are important for its construction. These dimensions are needed for PRA reconstruction, because the location of the recognized neighbours symbols has to be confronted with the physical ones.

The recognition of the types of symbols makes it possible to recover portions of the PRA that are then inspected until a complete window is found. Then the object and the face of the object that used this portion of PRA for encoding can be identified by simply querying the database that stores the mapping between the physical PRA and the object's faces (see next chapter).

The geometrical dimensions of the grid of symbols are used for POSE recovery (see Chapter 5). This information can be further used by a robot to manipulate the objects. For some very specific application, like space stations or nuclear plants, we believe that this method of object recognition should be considered.

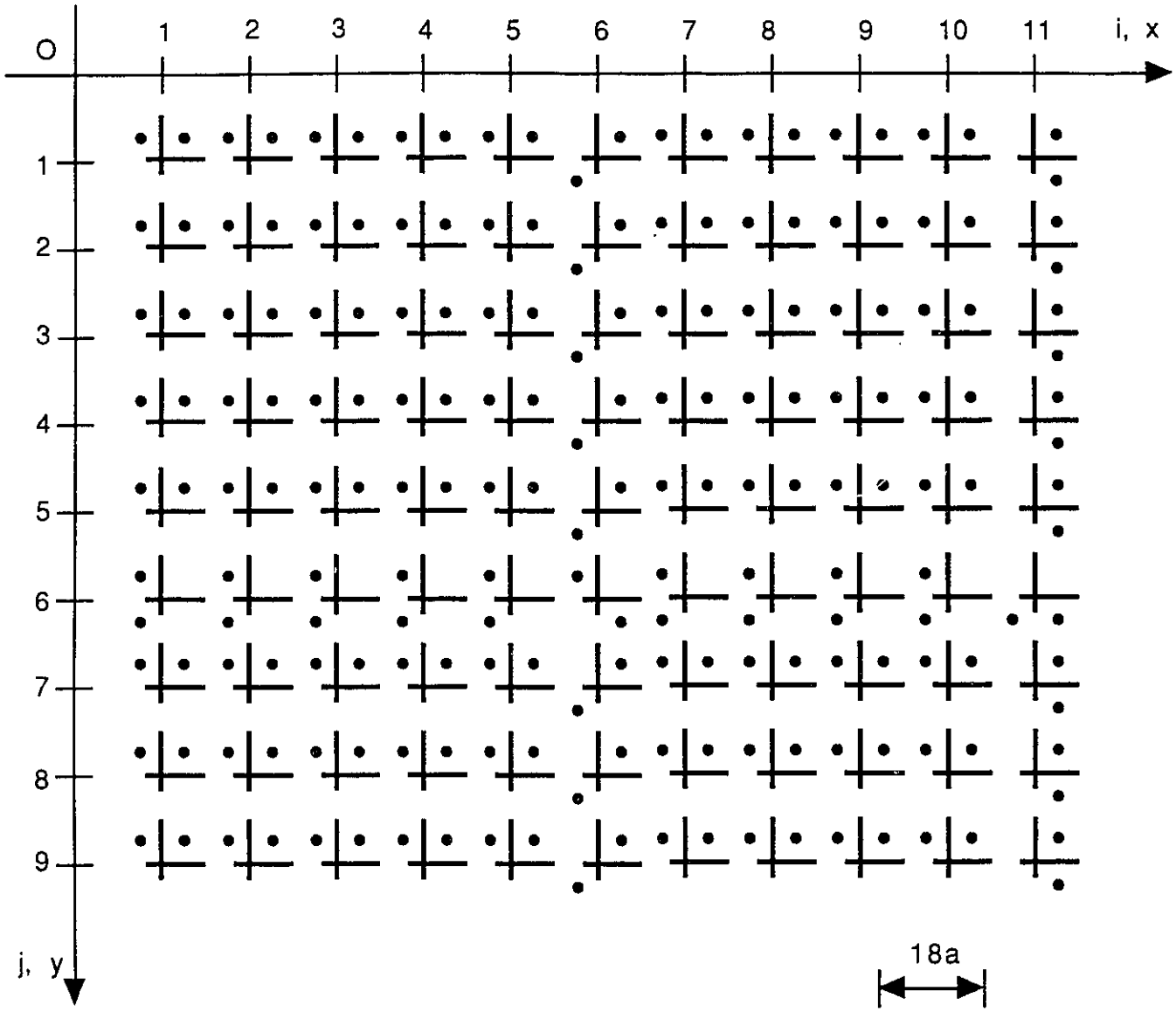


Figure 3.4. A physical PRA.

3.2. Image Processing for Symbol Recognition and Physical PRA Reconstruction

Next we present how the chosen symbols are recognized using image processing techniques. Any operational vision system, with this goal must have a hierarchical processing architecture: the *low-level processing* for feature extraction and the *high-level processing* for image interpretation and/or understanding [29].

3.2.1. Low-level processing

The symbol set presented in the previous section was designed in such a way as to make it possible to recognize the symbols using a skeleton approach. Thus, before applying the algorithm for symbol recognition, the skeleton of the image is required. We will start by presenting the low-level processing operations that will transform the original image, obtained from a camera, into a skeleton image ready for further processing.

The Threshold

A classical problem which must be faced by any symbol recognition process is the extraction of meaningful objects from the background of the pattern scene. This can be done by applying a threshold operation to the scene image.

Image thresholding is a technique for converting a grey-level image into a black and white image. Pixel values below a specified threshold are all converted to black, whereas pixel values at or above the threshold are converted to white.

Since the threshold operation is one that will influence dramatically the results of the symbol recognition procedure that we use, it is very important that it works well. The best way to pick the threshold, T, of a given image is to search the histogram of grey levels, assuming that it is bimodal, and find the minimum value between the two peaks. Finding the right minimum value between the peaks of a histogram can be difficult when the histogram is not a smooth function. Smoothing the histogram can help but does not guarantee that the correct minimum can be found.

The *single-threshold* method (*global threshold*) is useful in simple situations, but it cannot solve problems when the image has a background of varying grey levels, or when regions vary smoothly in grey level above and bellow the threshold.

A way to deal with the thresholding problem is to start the symbol recognition after the global thresholding of the image has been done by means of *histogram analysis*. If the recognition of a large enough patch fails due to illumination problems, the recognition procedure can be retried with a decreased or increased global threshold. The recognised symbols of the first try can obviously be preserved. Because the skeleton operation is very time consuming, we consider this approach unsuitable.

Two techniques can be applied to ameliorate this difficulty [30]:

- apply a *local threshold* with threshold values determined locally on square regions of the size of the symbols
- high-pass filter the image to deemphasize the low-frequency background variations and then apply the standard global threshold technique.

Local thresholding techniques have proved to be very robust and capable of handling real life situations where severe shading occurs, but the method is very slow. Apart from that it is very difficult to automatically chose the appropriate filter size for a local thresholding

technique for symbols that have different sizes, depending on the distance between the camera and the object, the focal length and the physical size of the symbols used to encode a given face of an object. We find this method inpractical for our purpose. With more processing power, this technique may prove to be the best for symbol recognition.

The second technique is the one that we used for thresholding. First a local maximum filter with size 5 by 5 is applied. After this operation an intermediate image is obtained in which the encoding symbols have disappeared but the general illumination effects (like shadows for instance) on the background are still present. Then, by extracting this intermediate image from the original grey scale image, an enhanced image is produced in which the encoding symbols (represented by dark pixels) appear on a uniformly bright background which is not affected by negative illumination effects. This final image is now suitable for global thresholding.

The Skeleton

A skeleton transform is applied to the resulting image. The 1-bit thick skeleton is an information preserving representation of the symbol shapes [30]. Despite its noise sensitivity, the skeleton method is used for the symbol recognition application since vertex-points and end-points can be easily recognized in a 3-by-3 pixel neighbourhood. As we will see the encoding symbols are recognized by their number of end-points and vertices in the skeletonized binary image.

Due to the fact that the skeleton operation can give errors if cracks and breaks appear in the symbol image, a closure operation (see [30] and [31]) with a size of one pixel and 8-neighbourhood is applied.

The algorithm used to produce the skeleton is based on the Hilditch crossing number, [31], defined as:

$$X(p) = \sum_{i=0}^{3} n_{2i} \vee (n_{2i+1} \wedge n_{2i+2})$$

where Σ is a modulo-8 arithmetic sum, $\wedge$ is the logical operation AND, $\vee$ is the logical operation OR, and $\mathbf{n_0}$ to $\mathbf{n_8}$ are the binary values of the eight neighbours of the current pixel **p** as shown in Table 3.1.

As a function of their crossing number, the binary image pixels can be classified as:

- isolated pixels characterized by X(p)=0
- break pixels characterized by X(p)>1
- contour pixels , other than break pixels characterized by X(p)=1.

n_2	n_1	n_0/n_8
n_3	p	n_7
n_4	n_5	n_6

Table 3.1. The neighbour pixels for Hilditch crossing number calculation.

By deleting all pixels that have a Hilditch crossing number greater than 1, and repeating this thinning operation until no deleting occurs, an 1-pixel thick skeleton symbol is obtained that still preserves the needed information for symbol recognition.

The obtained skeleton image is saved as an image with three bit planes. The first plane contains all points belonging to the resulting skeleton, so these pixels will have a grey value of 1. The end points, the points that have only one neighbouring pixel with a grey value equal to 1, are saved in the second bit plane. Thus, the resulting value for these pixels is $2^0+2^1=3$. Finally, the third plane will contain only the vertex pixels, the pixels that have more than 2 neighbouring pixel that have a grey value of 1, will have a grey value $2^0+2^2=5$.

Because the skeleton operation is very time consuming and because under normal circumstances just a few cycles are needed to transform any valid symbol to its skeleton representation, we restricted the number of thinning operations to four. (It makes no sense to continue the thinning operation only for the noise presented in the image.)

3.2.2. Symbol Pattern Recognition

The symbol recognition program starts by loading the skeleton image. As we have seen, the background of the image produced by the skeleton procedure has the grey value 0, and the skeleton of the symbol in the image consists of three kinds of pixels:

- *end pixels* - the points with one neighbour; they have a grey value of 3
- *vertex pixels* - points where three or more skeleton lines meet; they have a grey value of 5
- *link pixels* - the rest of the skeleton ; pixels which have a grey value 1.

Figure 3.5 shows the skeleton obtained from a valid symbol and which is the input for the symbol recognition program.

The main body of the symbols that form the grid contains a vertex pixel, that we consider to be the origin of the symbol, and four end pixels (see Figure 3.2).

Because the vertices are the most unique in the image the algorithm for symbol recognition starts with a linear scan of the image for these vertices. This is a very efficient way to start because the skeletons that don't contain vertices, and therefore cannot be skeletons of the main body of the symbols, are immediately rejected. This excludes the vast majority of the noise in the image from being evaluated and also all the dots that are part of a symbol, but not necessarily for the moment.

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	**3**	0	0	0	0	0	0	0	0	0	**3**	0	0
0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	**1**	**1**	0	0	0
0	0	0	0	0	0	0	0	**1**	0	0	0	0	**1**	**1**	0	0	0	0	0
0	0	0	0	0	0	0	0	0	**1**	0	**1**	**1**	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	**1**	**5**	**1**	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	**1**	**1**	0	**1**	0	0	0	**1**	0	0	0	0	0	0
0	0	0	0	0	**3**	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	**3**	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 3.5. The skeleton representation of the symbol "b".

The information about each found vertex is stored in the *vertex_info* structure which is defined as follows:

```
typedef struct vertex_info {
        unsigned char       val;
        int                 x_pos;
        int                 y_pos;
        struct line_info    *start_line_list;
        struct vertex_info  *next;
        };
```

The fields of this structure contains the following information:

- *val* - the grey value of the vertex -it has to be 5 under normal conditions
- *x_pos, y_pos*- the x and y pixel location of the found vertex
- **start_line_list* - a pointer to a list of lines that start in the vertex
- **next* - a pointer to a vertex_info structure that makes it possible for a list of vertices to be constructed. This is a very practical and efficient way to store related data items.

Next, starting from each vertex point that was previous detected and stored in a *vertex list*, the algorithm is looking for lines that start in the given vertex and a list of lines is constructed. The information about each line is stored in a *line_info* structure, defined as:

```
typedef struct line_info {
        int             x_start, y_start;
        int             x_end, y_end;
        unsigned char   start_value;
        unsigned char   end_value;
        float           length;
        struct line_info *next;
        };
```

The fields of this structure have the following meaning:

- *x_start* and *y_start* - the x and y pixel location of the beginning of the line
- *x_end* and *y_end* -the x and y pixel location of the end of the line
- *start_value* -the grey level value of the pixel that starts the line
- *end_value* -the grey level value of the pixel that ends the line
- *length* - the line length
- **next* -a pointer to a structure of the same type.

At this point the real symbol recognition process can start. For each vertex the *line_info* list is studied and a connectivity table is constructed. Every discovered line in the *line_info* list is given a line number, the initial one getting the number 0 (see Table 3.2 for an example of a connectivity table that corresponds with the skeleton in symbol given above).

Line nr.	x_start	y_start	start value	x_end	y_end	end value	length
0	112	79	5	120	75	3	8.94
1	112	79	5	114	87	3	8.24
2	112	79	5	108	81	3	4.47
3	112	79	5	110	75	3	4.47

Table 3.2. An example of a symbol connectivity table.

By analyzing this table, a decision is made if the given vertex belong to a valid symbol or not. This evaluation is done with the following constraints:

- each symbol has to contain exactly four lines
- each line has to start in the given vertex and end with a pixel that has a grey value 3 (an end point)

- exactly two groups of lines according to their length can be formed, each group containing exactly two lines.

A symbol list is constructed by taking all valid symbols that passed this evaluation test. The information about each symbol is stored in a structure, defined as:

```
typedef symbol_info {
        unsigned char       symbol_val;
        int                 x_org, y_org;
        float               direction;
        float               size;
        struct symbol_info  *left, *right, *top, *down;
        struct symbol_info  *next;
        };
```

At this point the following fields of the structure are already filled:

- *x_org* and *y_org* -x and y pixel coordinates of the origin of the symbol
- *direction* - the angle between the x axis of the symbol and x axis of the image
- *size*- the value of parameter a that defines the symbol size (see Figure 3.1). It is calculated by summing the length of all lines that form a symbol and dividing the result by 24
- **next* a pointer to the next found valid symbol.

To find the *symbol_val* that represents the symbol value taken from a subset of S defined in 3.1.1 the original image is reloaded and the four spots are checked for the dots that make the symbols differentiate between them. The positions of these spots are determined by doing simple rotations using the size of the symbol that is already known.

For the subset **S1** shown in Figure 3.3 just one dot has to be found for any symbol. The position of the spot will determine the value of the symbol (a number between 0 and 3). For the subset **S2** presented in the same figure, exactly two dots has to be found for each symbol and their position will uniquely determine the value of the symbol (a number between 0 and 5).

Explanations about how the remaining fields of this structure are filled is presented in the next section.

The results of the presented symbol recognition method suffer from three classes of errors. First, some symbol may not, for whatever reason, be recognized as being a symbol. This can easily happen using the skeleton method when a symbol is not properly thresholded and therefore appears to be "broken" in the binary image. These kind of errors are not normally disastrous. If enough symbols are recognized to provide a pseudo-random window, then the object can be identified. It is a serious problem only if a large portion of the grid is lost or if a PRA constructed using method 3 is used. For the object recognition to still be possible, a new image is required from a different angle and/or distance to the object.

Second, non symbol objects can be recognized as being symbols. This kind of error is not very common because after the first symbol evaluation the original image is reloaded and is not very probable to find a dot in the right spot. The errors that still occur can be rejected after the grid reconstruction procedure, when a new restriction can be imposed (each recognized symbol has to belong to the grid).

Finally, symbols can be recognized with the wrong value (for example a "1" can be recognized as being a "2"). This error can happen in the last stage of symbol evaluation procedure and occurs when some dots that do not belong to a symbol are found in the

right spot of the symbol. To reduce this kind of error the symbol subset is chosen to insure a Hamming distance D_{ij}=2 for **S1** and **S2**.

3.2.3. The PRA Reconstruction

The PRA reconstruction procedure is independent of the employed symbol shapes and the corresponding symbol recognition technique. Its input consists of sets of extracted symbol features, and it deals with their organization into a list of symbol_info structures and the recognition of 2-D grid pattern in which the PRA are arranged on the object faces.

When a symbol is discovered, its features are used to fill the fields of the symbol_info structure as presented above, and are then added to the linked list of symbols already recognized. Now the last four fields of the symbol_info structures have to be filled. The **top*, **down*, **left*, and **right* are pointers to four symbol_info structures that store the features of the four grid neighbours of a given recognized symbol. This is done by calculating the expected coordinates of the current neighbouring symbol (see Figure 3.6) and then comparing with the actual measured coordinates of these neighbours stored in the *x_org* and *y_org* fields of the symbol_info structure. The expected coordinates can be calculated by considering the specific design of the symbol (see Figure 3.2) and the grid (see Figure 3.4). For example, for the top neighbour the expected coordinates (x_exp, y_exp) are calculated using the following relations:

$$x\_exp = x\_org + 18a * \cos(\pi/2 + direction)$$
$$y\_exp = y\_org + 18a * \sin(\pi/2 + direction)$$

where: 18a is the grid step chosen when the PRA was designed

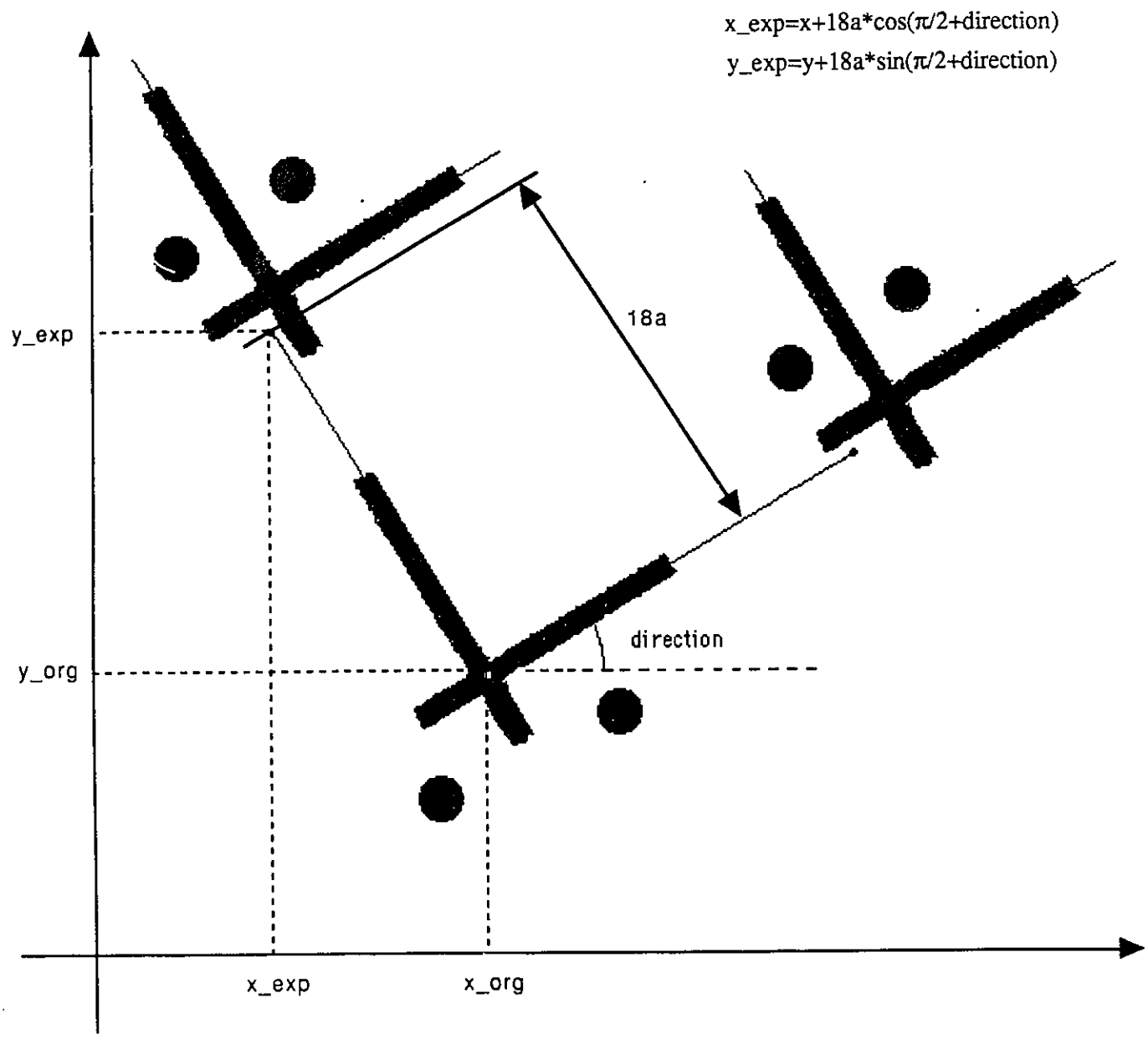


Figure 3.6. Calculation of the expected top-neighbour.

direction- is the angle between the x axes of the symbol and the image.

If the difference between the calculated coordinate values and the measured symbol coordinates does not exceed a certain threshold, the neighbouring candidate is accepted for integration into the PRA grid, and the appropriate pointer is set to it and the neighbour's corresponding opposite pointer is set to the newly added symbol (top to down, and left to right), as shown in Figure 3.6. In this way, erroneously recognized symbols are rejected before being integrated into the PRA. If more than one candidate is found for a given PRA grid node, then the "best fit" is chosen. In this way a graph is built that represents the perceived grid (see Figure 3.7).

The next step is to locate a pseudo-random window in the recognized grid. Because in our experiments we have constructed a PRA using the fourth method (see section 2.2.1), the missing symbols from the grid are not very critical for the success of window recovery. The *symbol_info* list is searched to find enough linked symbols in both the x and y direction. A pseudo-random window (PRW) in this case can have many "shapes" (see Figure 2.5), so it is quite probable to find a window and then to recover its (i,j) origin coordinates.

The last stage of the grid reconstruction and pseudo-random position recovery is an extended check of all symbols belonging to the recognized grid by comparing their values with the theoretical ones that can be generated using the method presented in chapter 2. At this stage it can be almost guaranted that only the good symbols are still present in the *symbol_info* list, and also that the recovered window contains only valid symbols.

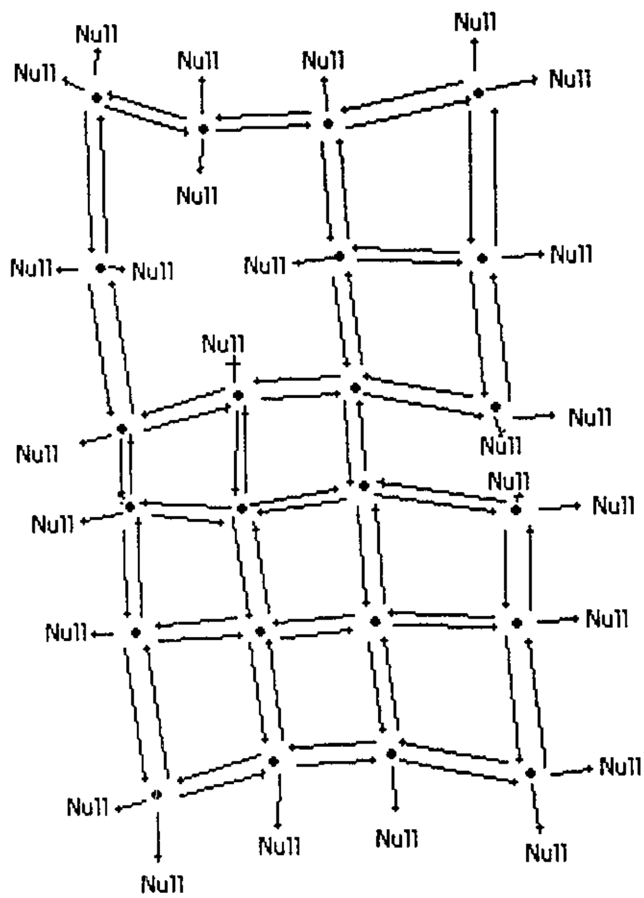


Figure 3.7. The PRA grid reconstruction.

PART 4

3-D OBJECT MODELS RELATIONAL DATABASE

The 3-D object recognition system requires a world model component that stores the object model data (see Figure 1.1). This component has to keep the "object model / PRA" mapping and the geometrical dimensions of the object models. Because this is the only information that the object-recognition system has about the world model objects, an object can be recognized if and only if it was previously stored here.

This chapter starts by presenting the mapping between the PRA and the object surfaces. Then a relational database that implements the world model component of the object recognition system is presented with an accent on the relational database scheme. Finally we describe the complex query of finding the face when the origin of a PRW is known.

4.1. "Object Model / P.R.A." Mapping

In order to keep the problem within manageable proportions, we restricted our attention to the case of polyhedral object models. These object models are easily described mathematically, their geometrical dimensions can be stored in a convenient way in a relational database, and always their surfaces can be flattened and unfolded to be mapped on the encoding PRA (see Figure 4.1). Because these objects have planar faces the symbol recognition procedure is more robust and a simpler POSE algorithm can be used.

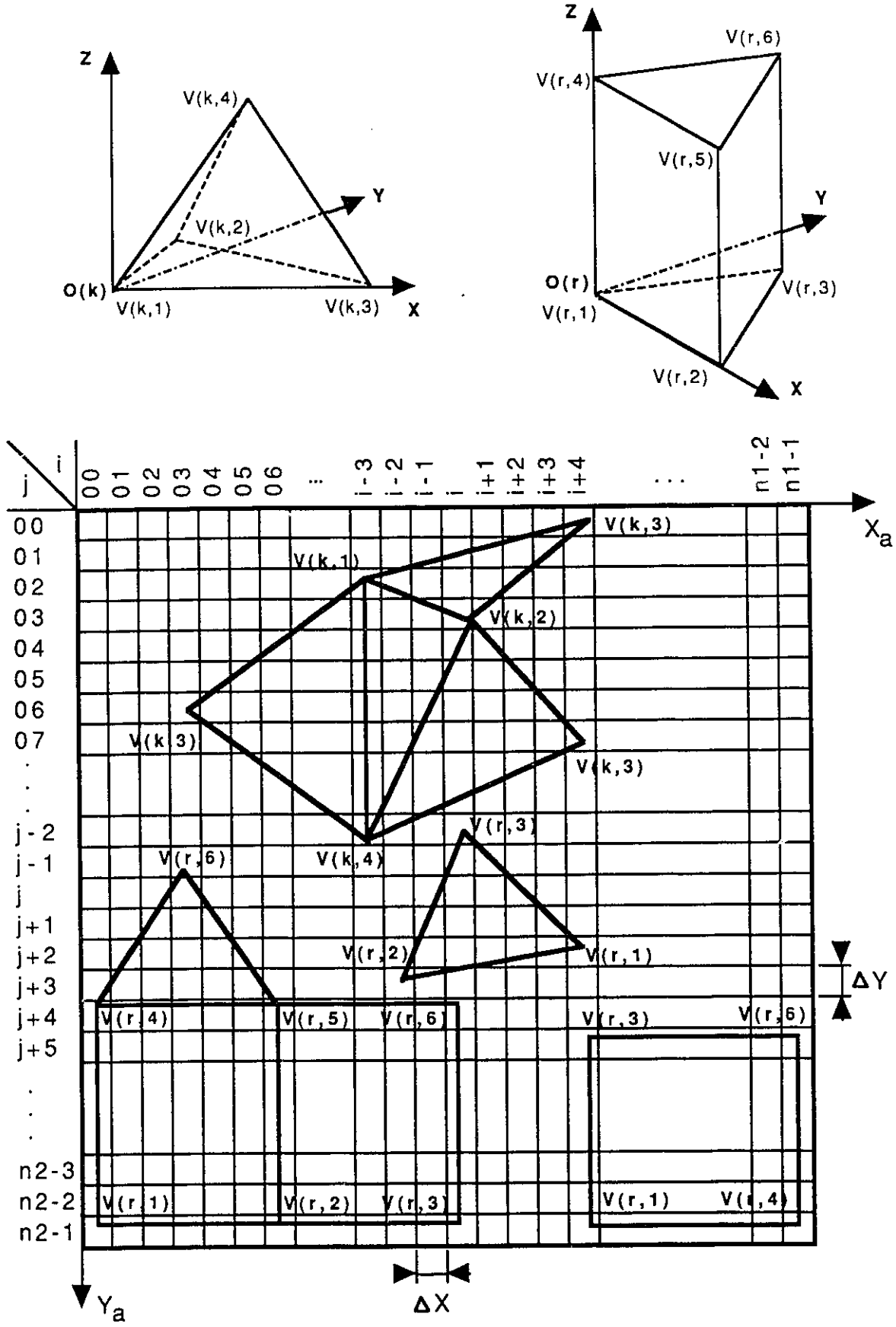


Figure 4.1. The geometrical models of two objects unfolded on the encoding PRA.

For physical objects that are not polyhedral, the closest polyhedral object model can be chosen to approximate the physical one (see Figure 4.2). Research has to be further conducted to find the best way to approximate physical objects and to efficiently store them in a relational database.

The visible surfaces of each object of interest are permanently encoded with elements of a large PRA in such a way that any region of the array is used only one time. The geometrical dimensions of the symbols and the grid step are chosen to satisfy the conditions presented in Chapter 3.1. Large surfaces are encoded with large symbols and small surfaces with smaller symbols. There always have to be enough symbols on a face to ensure that at least a PRW can be identified.

4.2. Object Database Design

A relational database, called the *Object Database*, has been chosen to store all the information that the object recognition system requires to perform its task. This database ensures an easy way to keep track of all the objects that the system can recognize and makes it possible for new objects to be stored or deleted. This is one of the features that makes such a system efficient.

The *Object Database* has been implemented using the C-library of functions given in [34].

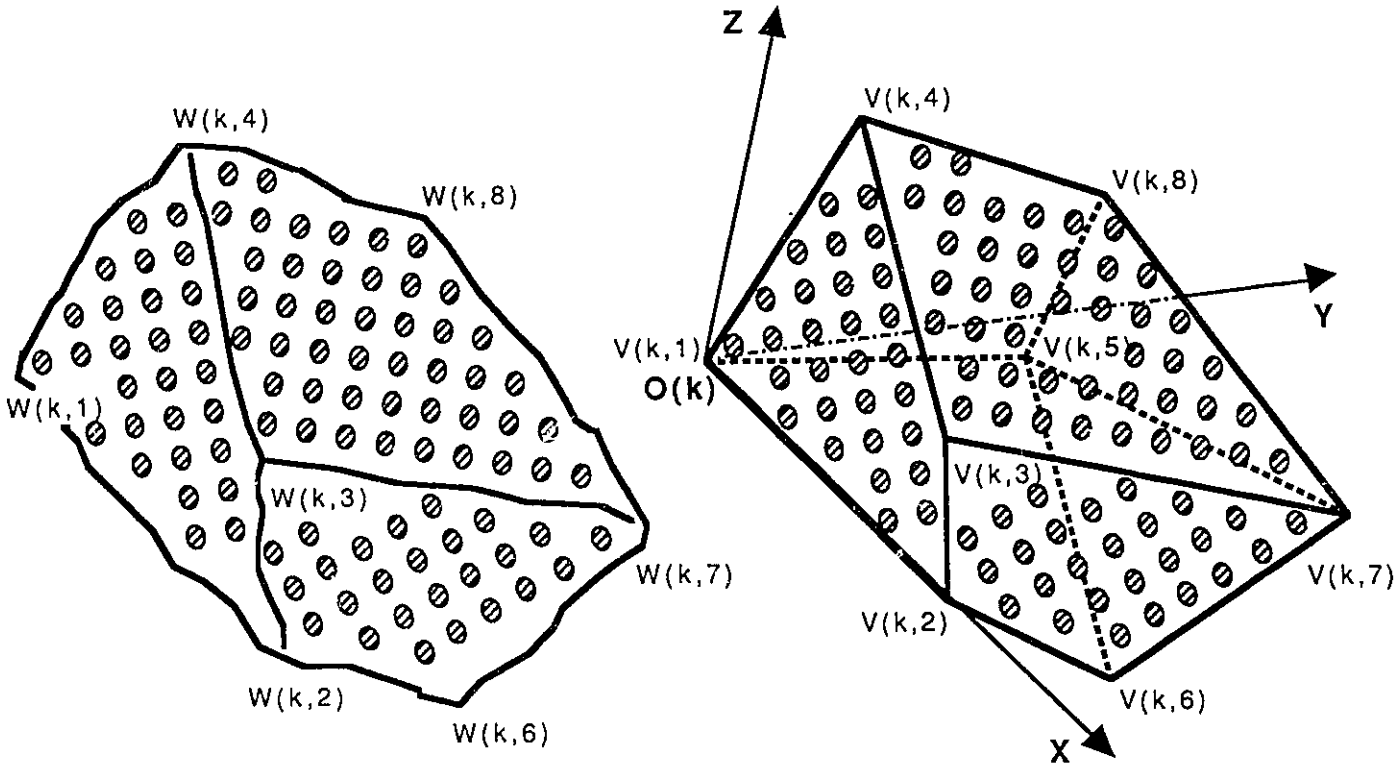


Figure 4.2. An object encoded with PRA elements and its polyhedral geometrical model with the attached 3-D reference system.

4.2.1. Object Database Specifications

The database was designed to provide the following functional and performance specifications:

Functional Specifications:

- The database will record and report the 3-D objects by *object_id* and *object_name*.

- The system will record and report the coordinates of each object's vertex in both the object's model frame and PRA coordinates. The PRA coordinates are necessary for object recognition and the geometrical dimensions of the objects are used for POSE recovery and defining the objects.

- The system will record and report the origin of each object's model frame and the coordinate system associated with each object, and also the origin of each face and the coordinate system associated with each face.

- The system will report the object and the face of the object when the (i,j) coordinates of a PRW are recovered and also the grid step used for encoding this face.

- The system will report the x and y coordinate of any symbol relative to the face coordinate system if the (i,j) coordinates of the symbols are known in the PRA coordinate system. This information is needed in the POSE process (see Chapter 5).

Performance Specifications:

- support up to 999 different objects
- each object has to have less than 99 faces
- each face has to have less than 99 vertices
- the name of each object has to be a word having a maximum of 25 characters (ex. cube, triangle_prism, square_pyramid)
- the name of each face must have less than 20 characters (ex. square, rectangle)
- each face must have less than 99 edges
- retrieval of data about an object and the coordinates of the points needed for POSE recovery will be in response to the window recovery process and POSE recovery process respectively.

4.2.2. Data Element Dictionary

The *data element dictionary* is a table of data elements including the names, data types, and length of every data element in the *Object Database*. It is central to the design of the database. All data items that appear in data files, on screen (for user), or are used for object representation and POSE recovery, must be defined in the data element dictionary.

The size of the database depends on the size of each element that appears in the data

element dictionary. For example the three characters used to keep the **object_id** permit a maximum of 999 objects to be stored by the Object Database.

Table 3.1 presents the data element dictionary for the *Object Database*.

In this data element dictionary:

- **object_id**, **vertex_id**, **coord_id**, **face_id**, and **edge_id** are primary keys or part of some primary keys of the relations in the database

- *object_name* and *face_name* are an array of characters used to give meaningful names to objects and faces

- *x_rel* and *y_rel* are the x and y coordinate in the face coordinate system relative to the face origin. There is always a vertex that has *x_rel*=0 and *y_rel*=0 that is the origin of the face and a vertex that has *x_rel*>0 and *y_rel*=0 that will give the xx' axis of the face coordinate system

- *X_coord*, *Y_coord*, and *Z_coord* are coordinates in the object's model frame of the vertices of the object model. A vertex with coordinates (0,0,0) always exists and is considered to be the origin of the object coordinate system. Another vertex (x,0,0) that always exists defines uniquely the X axis of the object coordinate system. A variable number of other vertices that have Z coordinate zero define a face that belongs to XY plane and thus, the object coordinate system is completely determined

- *x_pos* and *y_pos* are the coordinates in the PRA coordinates system of the vertices of the object and keep track of the mapping process.

	Data Element	Type	Byte
01	**object_id**	char	3
02	*object_name*	char	25
03	*number_of_faces*	char	2
04	*number_of_vertices*	char	2
05	**vertex_id**	char	2
06	*X_coord*	float	7
07	*Y_coord*	float	7
08	*Z_coord*	float	7
09	**coord_id**	char	2
10	*x_pos*	float	7
11	*y_pos*	float	7
12	*x_rel*	float	7
13	*y_rel*	float	7
14	**face_id**	char	2
15	*face_name*	char	20
16	*nr_of_edges*	char	2
17	*face_surface*	float	7
18	*grid_step*	float	7
19	**edge_id**	char	2
20	*lengh_of_edge*	flaot	7

Table 4.1. The Object Database data element dictionary.

4.2.3. The Object Database Scheme

A schema is the expression of the database in terms of the files it stores, the data elements in each file, the key data elements used for record identification, and the relationships between files. We present next the *Object Database* Scheme:

OBJECT_DATABASE SCHEMA:

OBJECTS: **object_id**, *object_name*, *number_of_vertices*,*number_of_faces*;

VERTICES: **vertex_id, object_id**, *X_coord*, *Y_coord*, *Z_coord*;

PLANAR **coord_id, vertex_id**, *x_pos*, *y_pos*, *x_rel*, *y_rel*;

FACES: **face_id, object_id**, *face_name*, *nr_of_edges*, *face_surface*, *grid_step*;

EDGES: **edge_id, face_id**, *length_of_edge*;

F_TO_V: **vertex_id, face_id;**

E_TO_V: **vertex_id, edge_id;**

The *Object Database* contains the following files: OBJECTS, VERTICES, PLANAR, FACES, EDGES, F_TO_V, and E_TO_V. The F_TO_V and E_TO_V are connector files and they were introduced to change the many-to-many relationship between FACES and VERTICES, and EDGES and VERTICES to one-to-many relationships.

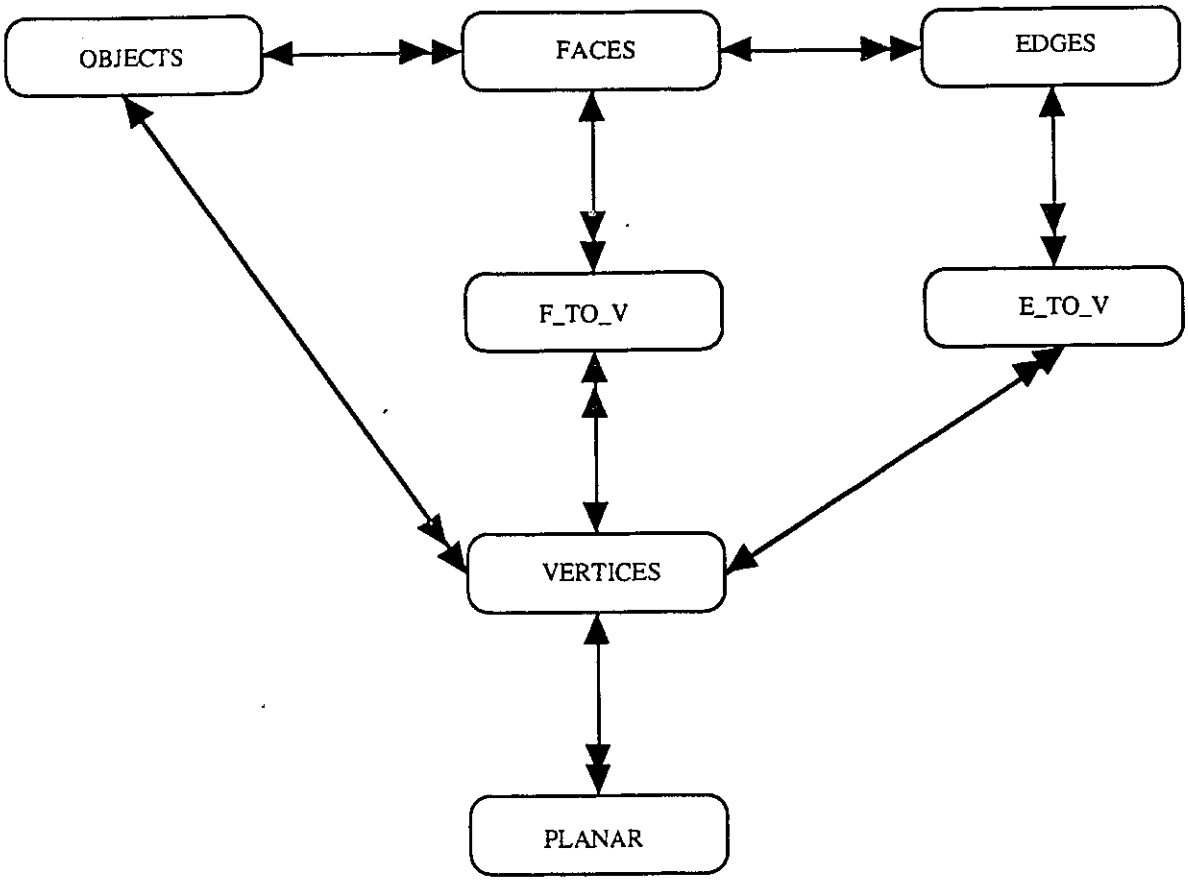


Figure 4.3. The file relationships for the Object Database.

A many-to-many relationship between two files changes to two, one-to-many relationships if a new file is created in the database. Figure 4.3 shows the *Object Database* files and the relationships between them.

The one-to-many relationships are implemented by including in the primary key of a file the primary key of the other file, that is, in one-to-many relation with the first one. The final form of the *Object Database* Scheme contains only one-to-many relationships.

4.2.4. The Database Management System (DBMS)

A DBMS is used to manage a database. It is placed between the applications programs wanting to get at the data records and the database waiting to be processed and has to pass data back and forth between these two components. It contains the following components (see Figure 4.4):

1.) *Data Definition Language* (DDL) provides a description of the database in a format that the DBMS can interpret. The schema that was previous designed has to be expressed using the DDL. Various techniques exists for writing the DDL. The C language was used to express the DDL for the *Object Database.*

The data model that the DBMS supports has to be considered when writing the DDL. The relational model used for Object Database requires that the files, the search keys, the attributes of each record, and relationships between files be sent to the DBMS through the DDL. The object.c and object.h C source code files shows how the *Object Database* schema is expressed in the C language to be interpreted by the DBMS (see Appendix A3 for the DDL for *Object Database*).

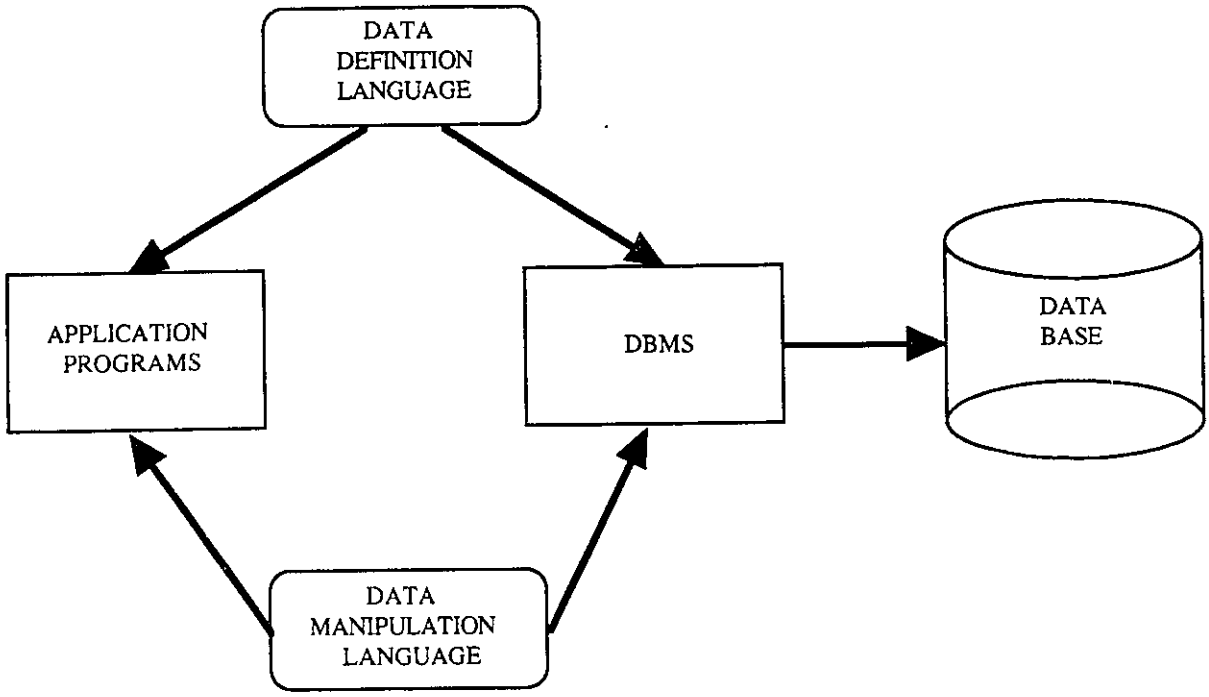


Figure 4.4. The Object Database components.

Because the relational data model does not provide a way to enforce relationships, the integrity of interfile relationships is the responsibility of the application programs.

2.) *Data Manipulation Language* (DML)-a corresponding language for programs to use so they can communicate with the DBMS. It provides an interface to the database. The DML uses external functions calls that are in fact C language functions (over 60).

These functions include functions that retrieve records from files by key search, add records to file, delete records from files, display records etc. Some of them are called directly from the C library functions and some were especially developed for the *Object Database* (see Chapter 4.2.5).

3.) *DBMS Utility Programs*- the software packages to support the management of data in the database. The library of utility programs includes: a data entry program, a query package, a sort program etc. The two specific features of *Object Database*, to retrieve a face of an object when an PRA vertex is known and to provide the coordinates of any PRA vertex in the object model frame, are part of the Utility Programs that were especially developed for this purpose.

The relational data model for *Object Database* is supported by inverted indexes into data files. The inverted index process use B-tree algorithms, a balanced tree of key values, to locate the data file record that matches a specified key argument. The index files have the same name as the data files, but the number of these index files increases with the number of attributes that form the primary key for a given relationship. The following names are valid index file names: OBJECTS.X01, FACES.X03, EDGES.X04, etc. Because of these files the size of the database is greater than the size of the data files that it contains.

4.2.5. Two utility programs

Two very important queries implemented in the *Object Database* as utility programs give the answer to the object recognition process and respective constitutes the input to the POSE recovery process. Next we give a short description of these queries.

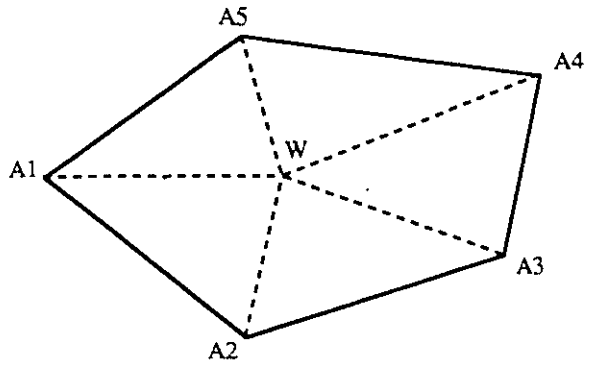
How object recognition query works

To implement this query (*obj_rec*) we use the following simple algorithm to identify if a point belongs to a convex polygon. Let W(i,j) be the origin of a PRA window that was identified by the symbol recognition and grid recovery process presented in Chapter 3. If the sum of the areas of all triangles formed by the W(i,j) vertex and each edge of an object face is equal to the area of the same object face, then the W(i,j) belongs to this face. If the sum of these areas is greater then the area of the object face then W(i,j) does not belong to this face (see Figure 4.5).

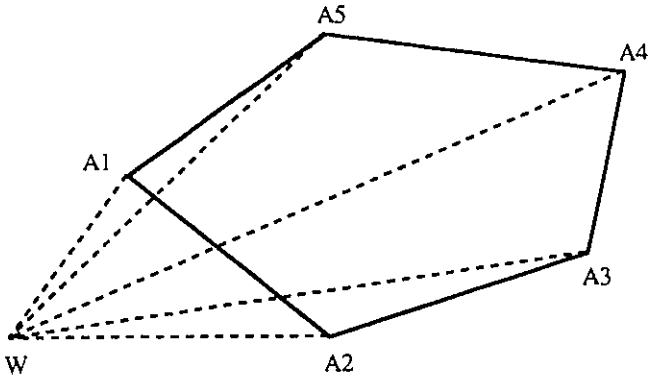
Using this algorithm and using simple queries for retrieving the face surface and the length of edges that are already implemented in the DML the object and the face that contain a given PRA index can be easily implemented. This is in fact the answer to the object recognition process.

How x, y coordinates of a given vertex are calculated

For POSE recovery discussed in Chapter 5, the *x,y* coordinates relative to the face coordinates system are needed. These coordinates are calculated by an utility program (*cal_cor*). The program gets as input the (i,j) coordinates of a given PRA vertex and a



W-belongs to face A1A2A3A4A5.



W-does not belog to the face A1A2A3A4A5.

Figure 4.5. How the face containing a given vertex is identified.

face which contains this vertex (the **face_id** search key). The inputs of the POSE recovery process are the coordinates of four vertices belonging to the same face of an object as we will see in Chapter 5.

PART 5

Camera Calibration for "POSE" Parameter Recovery

Each time a window is recovered it is possible to identify the face of the object which the window's origin point W(*i,j*) belongs to, as well as, to find the coordinates of all the recognized symbols relative to the object's frame by simply querying the Object Database.

The coordinates of the recognized symbols have to be known in both the image frame and in the object's model frame in order to determine the position, orientation and scaling estimation (POSE) of the camera relative to the object's model frame. This problem is also known as the problem of determining the elements of exterior camera orientation, or the camera calibration problem [35] or image-to-database correspondence problem [15].

The POSE problem can be stated as:

> *Given a set of m control points (the origin of m symbols in the grid used to encode a face of an object), whose 3-dimensional coordinates are known in the object's model frame, and given an image in which these m points are visible, determine the location of the camera (relative to the object's model frame) from where the image was taken* (see Figure 5.1).

Let us consider m control points Q_1, Q_2, Q_3, ... , Q_m whose coordinates are known in the object's model frame :

$Q_i(X_i, Y_i, Z_i)$ for i=1 to m.

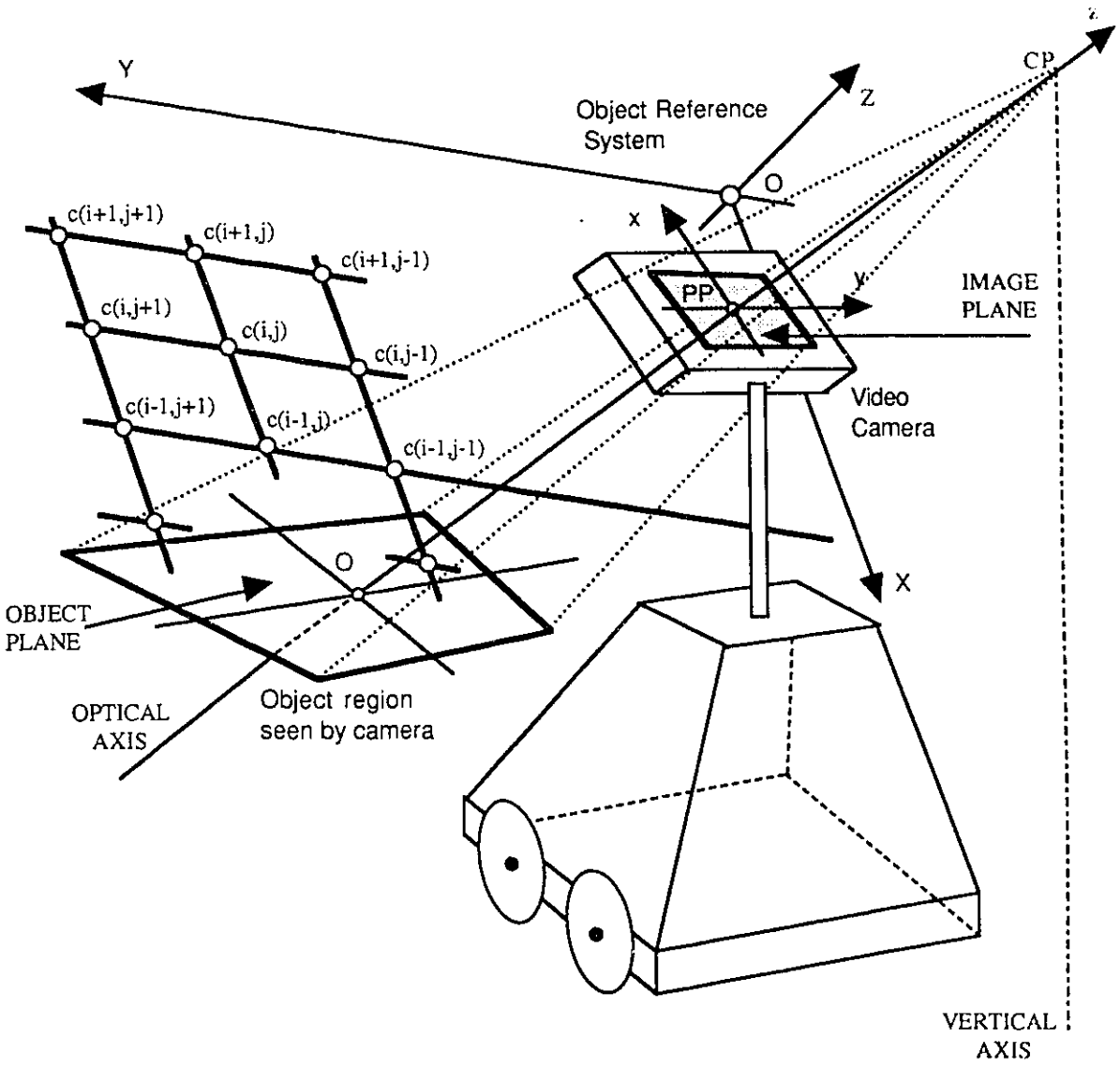


Figure 5.1. The POSE problem.

Corresponding to each point Q_i an image point P_i exists whose coordinates in the image frame are determined using image processing techniques. Let us suppose that these points are given by:

$$P_i(x_i, y_i, 0) \qquad \text{for } i=1 \text{ to } m.$$

The *optical axis* of the image is perpendicular to the image plane in the centre of the image and is identical with the z_{im} axis. For this reason the centre of the image coordinate system is also called the *principal point* (PPI). The *focal length* f defines another point on the optical axis the centre of perspective C_p, such that:

$$f=| C_pO_{im}|$$

First we are interested in finding how many control points are necessary for a unique solution for the position of C_p. It is known [15], [35] that for one and two points there are an infinite number of solutions, and a minimum of three points are required for a finite number of solutions to exist. Three arbitrary control points will yield four solutions. A unique solution can be obtained using four coplanar points or a minimum of six arbitrary control points.

This unique solution for six points can be obtained by determining the 12 coefficients of the 3-by-4 matrix T that specifies the mapping (in homogeneous coordinates) from the 3-D object's model frame to 2-D image frame:

$$[tx_{P_i}, ty_{P_i}, t]^T = \begin{bmatrix} C_{11} & C_{12} & C_{13} & C_{14} \\ C_{21} & C_{22} & C_{23} & C_{24} \\ C_{31} & C_{32} & C_{33} & C_{34} \end{bmatrix} [X_{Qi}, Y_{Qi}, Z_{Qi}, 1]^T$$

for i=1 to 6. This 3-by-4 transformation matrix (*camera matrix*) contains the three position and three orientation parameters of the image frame POSE relative to the object's model frame, and five camera parameters (focal length f, the X_{oim} and Y_{oim} coordinates of the principal point, and two lens distortion parameters). Due to the homogeneous coordinates, the overall scaling can be arbitrarily set $C_{34}=1$.

Each of these six correspondences provides three new equations but introduces one additional unknown (the homogeneous coordinate scale factor). Thus, for six control points an 18 linear equations system has to be solved for finding the 18 unknowns. Such an approach is not suitable because it is computationally expensive.

An analytical solution for the POSE problem called the Perspective-4-Point Problem (P4P) with all four control points lying in a common plane (a face of a polyhedral object) has been adopted as a more convenient approach for our case.

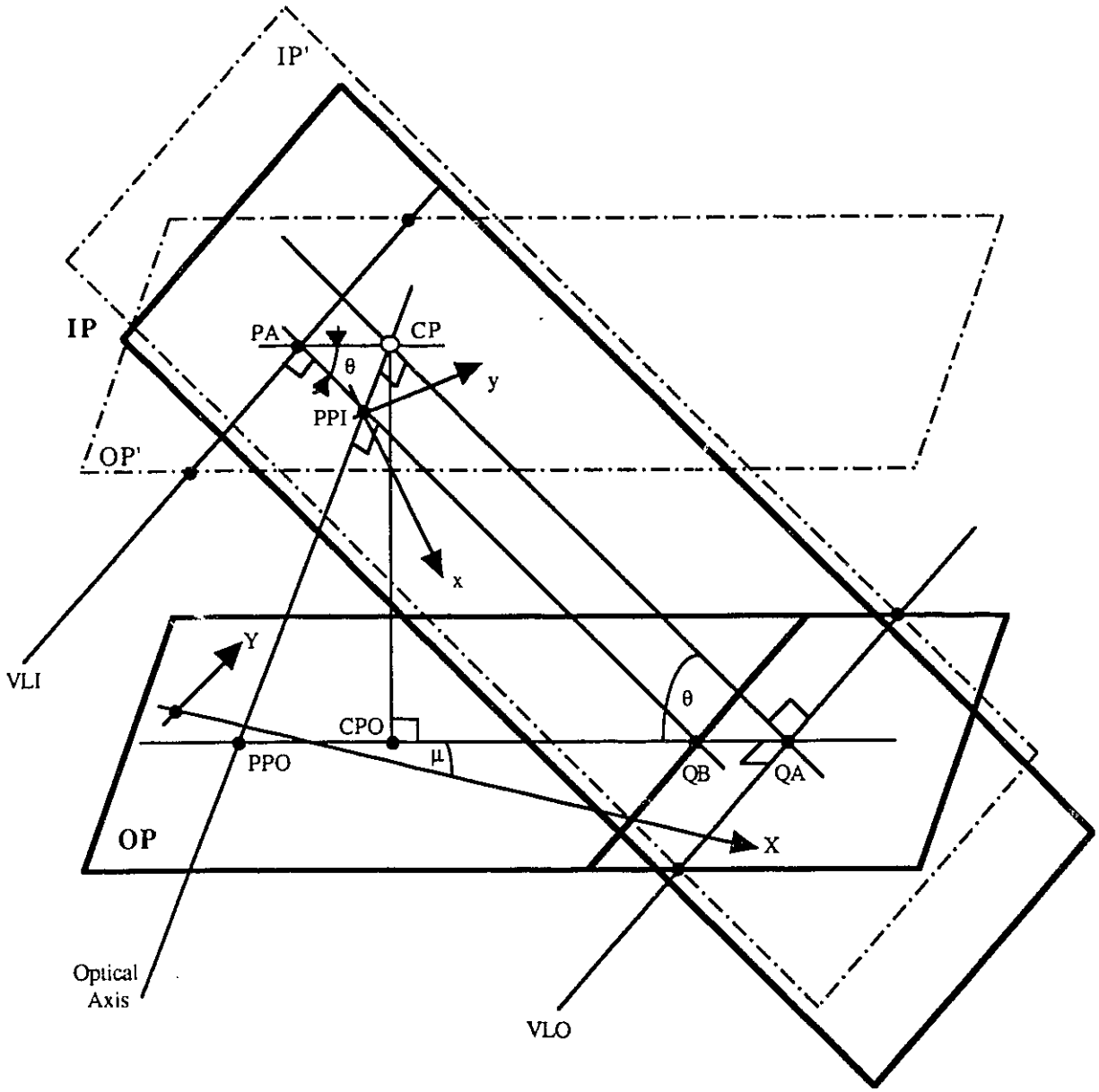
The P4P problem can be stated as follows [15] (see Figure 5.2):

Given:

- a correspondence between four points ($\mathbf{P}_i$(X_i, Y_i, 0), for i=1 to 4) lying in a plane in 3-D space (the object plane), and four points ($\mathbf{Q}_i$(x_i, y_i, 0), for i=1...4) lying in a distinct plane (the image plane)
- the distance between the centre of perspective and the image plane(the focal length **f** of the image system)
- the principal point of the image plane PPI

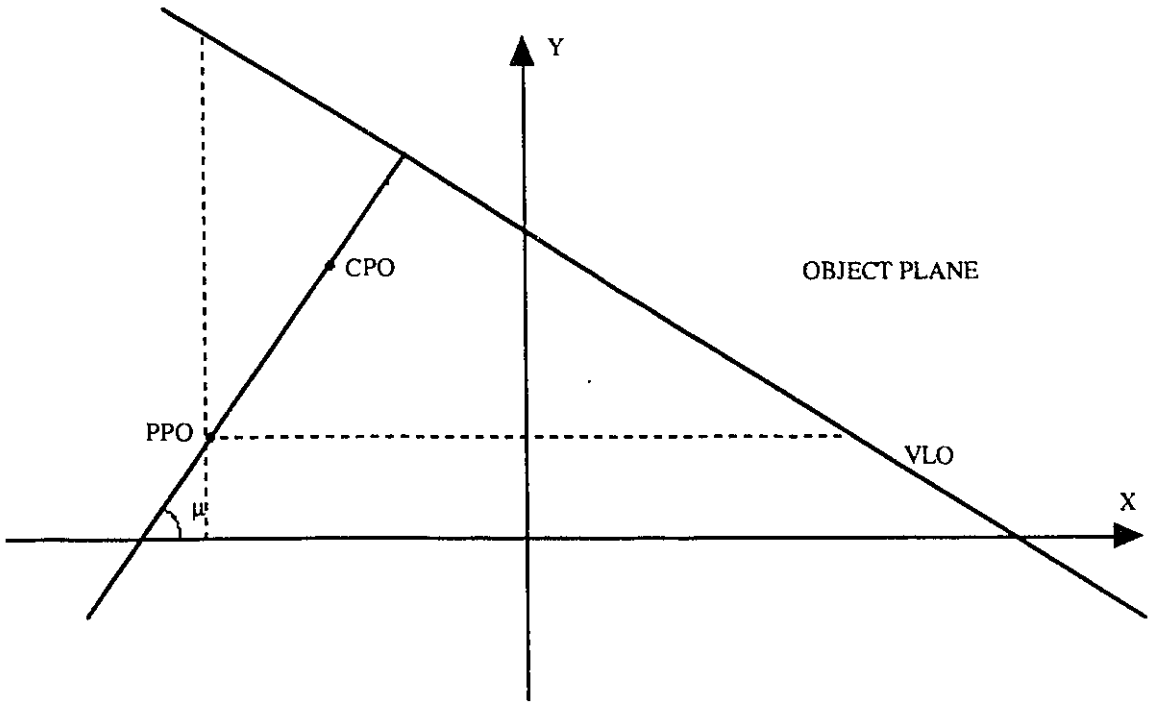
Find:

- the location of the centre of perspective relative to the coordinate system of the object plane.

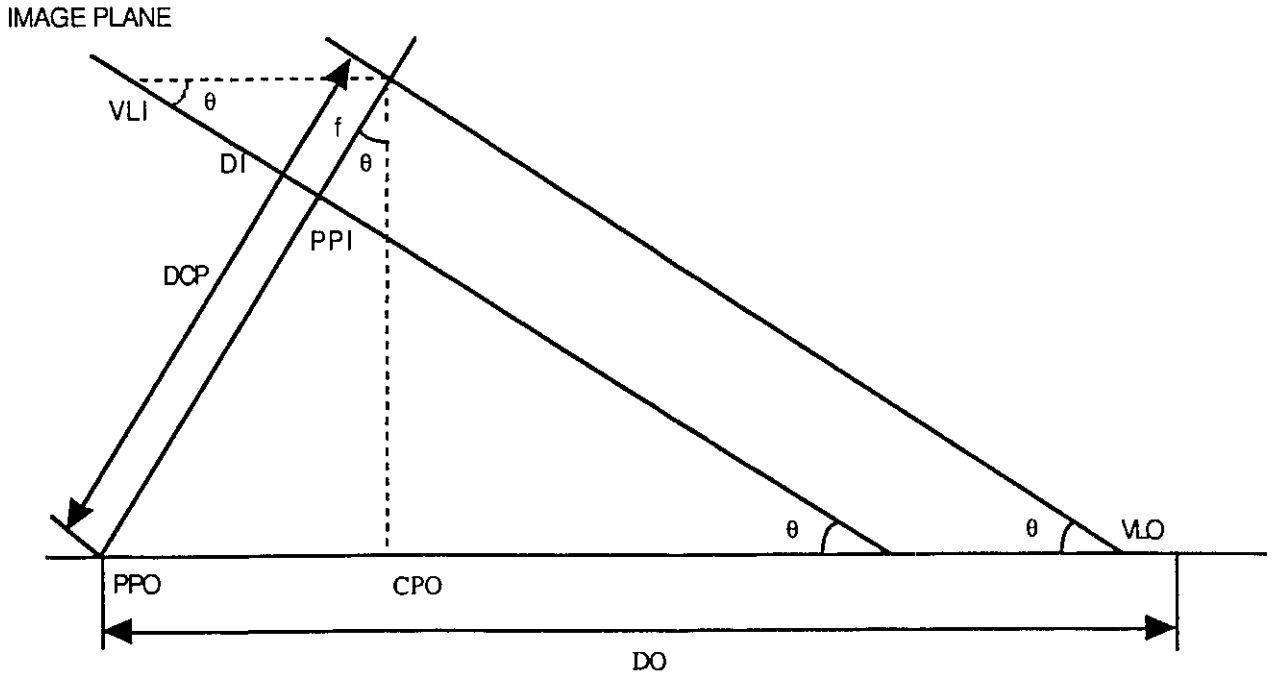


IP =Image Plane
OP =Object Plane
VLI =Vanishing Line in Image plane
VLO =Vanishing Line in Object plane
CP =Center of Perspective
CPO =Center of Perspective projection in Object plane
PPI =Principal Point in Image plane
PPO =Principal Point in Object plane

Figure 5.2. a) The P4P problem.



b)



c)

Figure 5.2. b) and c) The P4P problem.

Notation:

- P[3][4] -the image point matrix
- Q[3][4] -the object point matrix
- PPI -the origin of the image and the principal point of the image;
- the object coordinate system has the equation Z=0 in the reference coordinate system. Standard techniques are available to transform from this coordinate system into the object coordinate system [36]
- homogeneous coordinates will be assumed
- primed symbols represent transposed matrix

Solution:

The following ten step algorithm can be used to solve this specific P-4-P problem [2]:

Step 1

Determine the T[3][3] collineation matrix which maps points from the object plane to image plane:

$$[Pi]=[T]*[Qi]$$

Note. This matrix can be determined if at least four pairs of coplanar points are known in the object's model frame and respectively in the image frame.

Step 2

Determine the vanishing line in the image plane VLI[3] by mapping the ideal line in the object plane [0, 0, 1] using the transformation;

$$[VLI]=[inv[T]]' * [0, 0, 1]' =[a1, a2, a3]$$

The vanishing line in the image plane is a line given by the intersection between the image plane and an imaginary plane parallel with the object plane and containing the Centre of Perspective CP.

Step 3

Determine the distance DI from the origin of the image plane, which is also the principal point of the image plane (PPI) to the vanishing line VLI;

$$DI = \frac{a_3}{\sqrt{a_1^2 + a_2^2}}$$

Step 4

Determine the dihedral (tilt) angle μ between the image plane and object plane (the focal length **f** can be determined using a National Instruments Data Acquisition board and a feedback from the camera);

$$\mu = \text{arctg}(f/DI)$$

Step 5

Determine the vanishing line in the object plane VLO by mapping the ideal line in the image plane [0, 0, 1] using the transformation:

$$[VLO] = [T]' * [0, 0, 1]' = [b1, b2, b3]$$

Step 6

Determine the location of PPO in the object plane. The PPO is the point at which the optical axis of the lens pierces the object plane;

$$[PPO] = [\text{inv}[T]]' * [0, 0, 1]' = [c1, c2, c3]$$

Step 7

Determine the distance DO from PPO to the vanishing line VLO in the object plane:

$$DO = \frac{b_1 * c_1 + b_2 * c_2 + b_3 * c_3}{C_3 \sqrt{b_1^2 + b_2^2}}$$

Step 8

Determine the "pan" angle θ as the angle between the normal to VLO=[b_1,b_2,b_3] and the X axis in the object plane:

$$\theta=\text{arctg}(-b2/b1)$$

Step 9

Determine XSGN and YSGN. If a line (parallel to the X axis in the object plane) through PPO intersects VLO to the right of PPO, then XSGN=1. Otherwise XSGN=-1. Similarly for YSGN. Thus,

$$if\frac{b_1*c_1+b_2*c_2+b_3*c_3}{b_1*c_3}<0$$

then XSGN=1, otherwise XSGN=-1.

Similarly,

$$if\frac{b_1*c_1+b_2*c_2+b_3*c_3}{b_2*c_3}<0$$

then YSGX=1, otherwise YSGN=-1.

Step 10

Determine the location of CP(XCP, YCP, ZCP in the object plane coordinate system:

$$DCP=DO*\sin(\mu)$$
$$XCP=XSGN*\text{abs}[DCP*\sin(\mu)*\cos(\theta)]+c1/c3$$
$$YCP=YSGN*\text{abs}[DCP*\sin(\mu)*\sin(\theta)]+c2/c3$$
$$ZCP=DCP*\cos(\mu)$$

A C program that implements this method was developed and tested. The source code of the program is given in Appendix A4. The user has to input the coordinates of four recognized points and their corresponding image coordinates. Also, the focal length is a parameter that is not obtained automatically and has to be input by the user.

PART 6

EXPERIMENTAL RESULTS

Five representative types of 3-D objects were considered: cube, rectangular parallelepiped, triangular prism, square pyramid and triangular pyramid. Each encoded face of these objects was vertex mapped to a PRA that was constructed using method 4 presented in Chapter 2.2.1.

Two PRSs generated over GF(3) and GF(2) respectively were used to encode the *i* and *j* coordinates of the PRA. These PRSs are automatically generated using a "PRS-generation program".

The outputs from the pseudo-random sequence generation program are:

1.) The sequence that is used to encode the *i* coordinate of the PRA:

```
To generate a pseudo-random sequence the following parameters
are required:
      - an integer number 'q' that is a prime or a power of
        a prime ;
        This number 'q' is expressed as a power 'm' of a
        prime number 'p'.
      - an integer 'n' that gives the degree of the
        polynomial that is used to generate the PRS over
        GF(q).
```

Choose these three parameters:

- Enter prime 'p' (2 or 3) 3
- Enter integer 'm'(1 or 2) 1
- Enter integer 'n'(1-9) 6

You have chosen to generate a pseudo-random sequence characterized by the following parameters:

- p=3, m=1, q=3, GF(3)(a pseudo-random three symbols sequence).
- n=6 (the length of this sequence is 728).

```
0 0 0 0 0 1 0 0 0 0 2 1 0 0 0 1 1 1 0 0 2 0 0 1 0 1 2 0 2 1 2
2 2 1 1 2 0 0 1 0 2 2 0 2 1 1 0 2 1 1 0 1 1 1 0 1 2 0 0 1 2 2
2 0 2 2 0 0 2 1 0 2 0 1 1 1 1 2 2 0 0 0 2 0 2 0 0 1 2 1 2 0 2
2 1 2 2 1 0 1 2 0 1 1 2 2 2 2 0 2 0 0 0 2 1 2 0 0 1 1 2 2 0 2
0 2 0 2 1 2 1 2 1 1 2 1 2 1 0 2 1 2 1 1 1 1 2 1 0 0 0 2 1 1 0
0 1 1 0 1 0 2 0 1 2 1 1 2 2 2 1 0 2 0 0 1 1 1 2 0 2 0 0 2 2 1
2 0 1 0 1 2 2 2 1 2 2 0 0 1 2 0 2 0 2 2 2 1 2 1 0 0 1 2 1 1 0
2 2 1 0 1 1 0 1 1 2 0 1 2 0 2 2 2 2 2 1 0 0 0 0 1 1 0 0 0 2 0
1 0 0 1 2 2 1 0 2 2 0 1 1 1 0 2 2 0 0 1 1 0 2 0 2 0 1 1 2 1 2
2 0 2 1 2 0 2 1 1 2 2 1 1 0 2 0 1 0 1 1 2 2 1 2 0 2 0 1 2 2 1
2 2 2 0 1 2 0 0 2 2 2 2 0 1 0 0 0 2 2 1 0 0 1 0 1 1 0 2 1 2 0
1 1 1 2 2 2 0 0 2 0 0 2 0 1 2 0 1 2 2 2 2 2 2 0 0 0 0 0 2 0 0
0 0 1 2 0 0 0 2 2 2 0 0 1 0 0 2 0 2 1 0 1 2 1 1 1 2 2 1 0 0 2
0 1 1 0 1 2 2 0 1 2 2 0 2 2 2 0 2 1 0 0 2 1 1 1 0 1 1 0 0 1 2
0 1 0 2 2 2 2 1 1 0 0 0 1 0 1 0 0 2 1 2 1 0 1 1 2 1 1 2 0 2 1
0 2 2 1 1 1 1 0 1 0 0 0 1 2 1 0 0 2 2 1 1 0 1 0 1 0 1 2 1 2 1
2 2 1 2 1 2 0 1 2 1 2 2 2 2 1 2 0 0 0 1 2 2 0 0 2 2 0 2 0 1 0
2 1 2 2 1 1 1 2 0 1 0 0 2 2 2 1 0 1 0 0 1 1 2 1 0 2 0 2 1 1 1
2 1 1 0 0 2 1 0 1 0 1 1 1 2 1 2 0 0 2 1 2 2 0 1 1 2 0 2 2 0 2
2 1 0 2 1 0 1 1 1 1 1 2 0 0 0 0 2 2 0 0 0 1 0 2 0 0 2 1 1 2 0
1 1 0 2 2 2 0 1 1 0 0 2 2 0 1 0 1 0 2 2 1 2 1 1 0 1 2 1 0 1 2
2 1 1 2 2 0 1 0 2 0 2 2 1 1 2 1 0 1 0 2 1 1 2 1 1 1 0 2 1 0 0
1 1 1 1 0 2 0 0 0 1 1 2 0 0 2 0 2 2 0 1 2 1 0 2 2 2 1 1 1 0 0
1 0 0 1 0 2 1 0 2 1 1 1 1 1 1
```

2.) The sequence that is used to encode the *j* coordinate of the PRA is generated in the same manner as the *i* coordinate:

Choose these three parameters:

- Enter prime 'p' (2 or 3) 2
- Enter integer 'm'(1, 2, 3, 4) 1
- Enter integer 'n'(1-14) 6

```
You have chosen to generate a pseudo-random sequence
characterized by the following parameters:
        -p=2, m=1, q=2, GF(2) (a pseudo-random binary
sequence).
        -n=6 (the length of this sequence is 63).
0 0 0 0 0 1 0 0 0 0 1 1 0 0 0 1 0 1 0 0 1 1 1 1 0 1 0 0 0 1 1
1 0 0 1 0 0 1 0 1 1 0 1 1 1 0 1 1 0 0 1 1 0 1 0 1 0 1 1 1 1 1
1
```

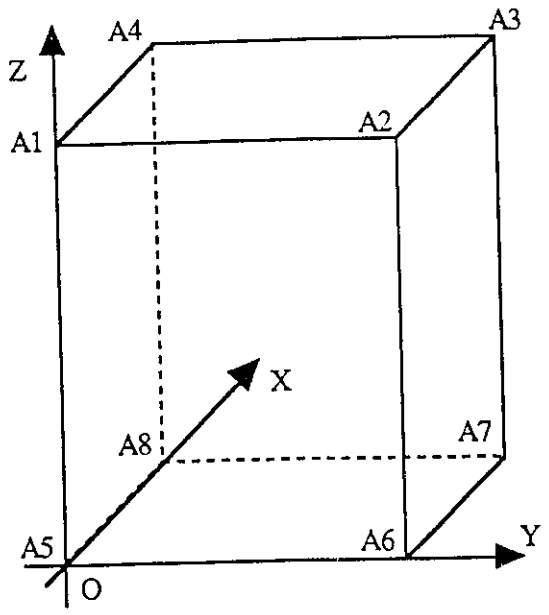
Using these two PRSs, a 728-by-63 PRA is obtained. To mark this PRA the 6-symbol set **S2** presented in Figure 3.3 is used. Table 6.1 gives the cross-product table used to encode this PRA.

	0	1	2
0	0	1	2
1	3	4	5

Table 6.1. The cross-product table for experimental PRA generation.

Figure 3.4 shows the top left corner of this array that is in fact used to encode the rectangular parallelepiped. Using portions of this array the faces of the objects were mapped and then stored in the Object Database. Figure 6.1 shows how the rectangular parallelepiped was mapped in the natural-code indexed 728-by-63 PRA and also the physical dimensions of these objects. The same technique was used to mapped the others four objects.

To test the object-recognition method, a vision system that uses a CCD camera and a Matrox Image 1280 image acquisition board resident in a 386 personal computer with a 25 MHz clock rate was used. The processing time for carrying out an experiment of recognition of an object is 12s.



	X[mm]	Y[mm]	Z[mm]
A1	0	0	150
A2	0	145	150
A3	110	145	150
A4	110	0	150
A5	0	0	0
A6	0	145	0
A7	110	145	0
A8	110	0	0

The coordinates of the rectangular parallelepiped relatively to its coordinate system

Geometrical Object Vertex	PRA (i,j) coordinates
A5-A1-A2	(000, 00)
A1-A2-A6	(010, 00)
A2-A6-A5	(010, 10)
A6-A5-A1	(000, 00)
A6-A2-A3	(022, 00)
A2-A3-A7	(028, 00)
A3-A7-A6	(028, 10)
A7-A6-A2	(022,10)
A7-A3-A4	(011, 00)
A3-A4-A8	(021, 00)
A4-A8-A7	(021, 10)
A8-A7-A3	(011, 10)
A8-A4-A1	(029, 00)
A4-A1-A5	(035, 00)
A1-A5-A8	(035, 10)
A5-A8-A4	(029, 10)
A2-A1-A4	(006, 11)
A1-A4-A3	(006, 21)
A4-A3-A2	(000, 21)
A3-A2-A1	(000, 11)
A8-A5-A6	(007, 11)
A5-A6-A7	(013, 11)
A6-A7-A8	(013, 21)
A7-A8-A5	(007, 21)

The mapping of the rectangular parallelepiped in the 728-by-63 PRA.

Figure 6.1.The rectangular parallelepiped mapped in the 628-by-63 PRA.

As was expected, the production of a skeleton is the most time consuming part. The results however are rather promising. With good illumination and a reasonable viewing angle, a grid containing about 50 symbols in a 256-by-256 image can be recovered. Extreme perspective distortion can result in wrong neighbour distance estimation.

By doing the image filtering, thresholding, and skeleton operation using the Image 1280 board the processing time decreased to 6s. This is because the skeleton operation is very time consuming when done on the PC, and also a lot of time is spent loading the images from the Image 1280 board into the main memory of the personal computer.

Since a small number of objects were used (only 5), the database search time was very small (milliseconds). Also the POSE recovery program does not require a large amount of time. A number of experiments were carried out on the previous described five objects with the conclusion that about 96% of the processing time is spent doing the symbol recognition and PRA reconstruction.

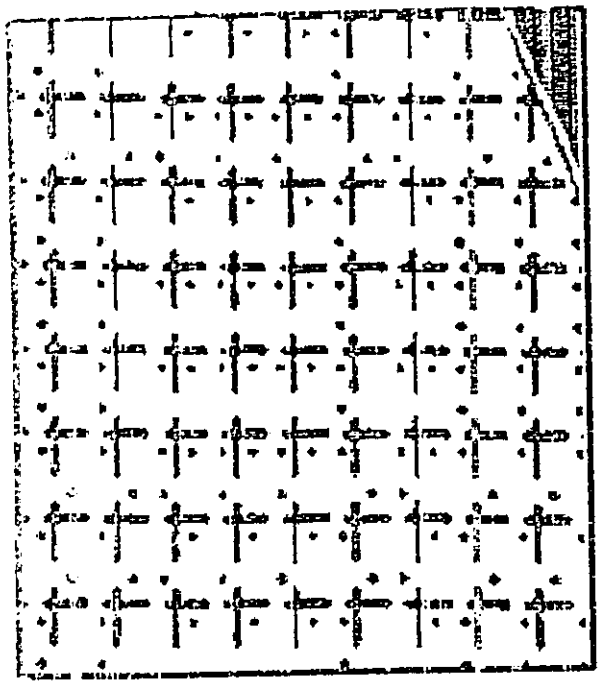
The described method appears to be robust in the sense that it never produced a grid with a false PRA encoding. Symbols may not be recognized due to some breaks in the skeleton and sometimes noise may be recognized as a symbol; however, during testing these falsely recognized symbols never had the correct size and orientation to be accepted and integrated as neighbours as required by the PRA reconstruction algorithm. The reconstructed grid may thus be incomplete, but the information which it contains is still reliable.

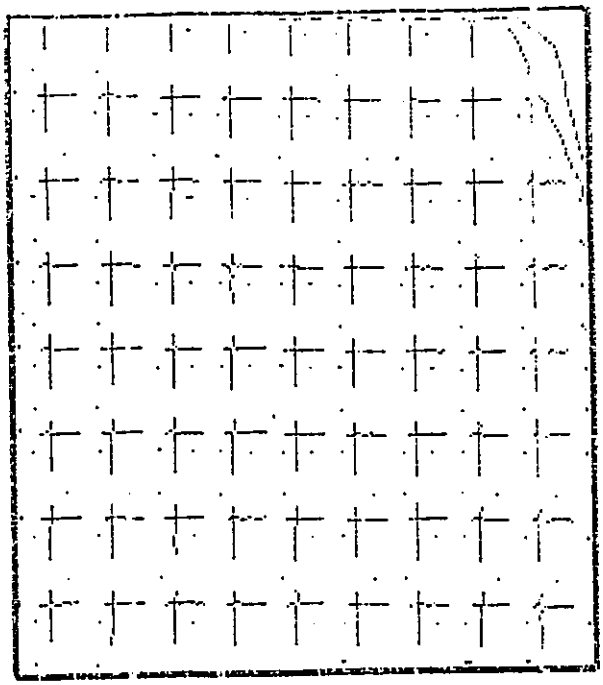
Figure 6.2, 6.3 and 6.4 show the output from the object recognition program for the triangular pyramid (two faces) and rectangular parallelepiped. Each figures contains

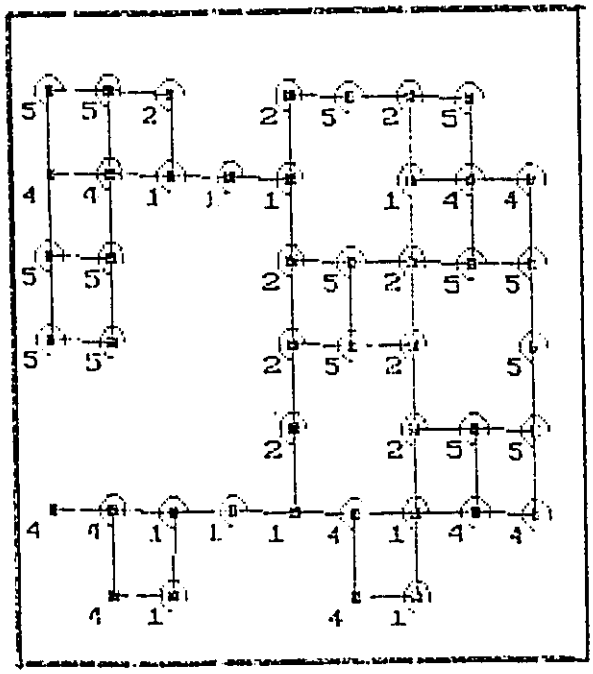
- the thresholded image of a face of an object
- the same image after the skeleton operation
- the grid reconstruction from the recognized symbols
- the corresponding object and face of this object after quering the database.

Because at this time the POSE recovery process is not integrated in the object-recognition system the inputs to this program had to be introduced by the user. For the rectangular parallelepiped the outputs of the POSE recovery process is presented in at the end of this chapter. The meaning of the parameters that appear in this output are given in Chapter 5 and also in the Appendix A4.

These experiments have validated the practical applicability of the PRA encoding for visual recognition of 3-D objects.



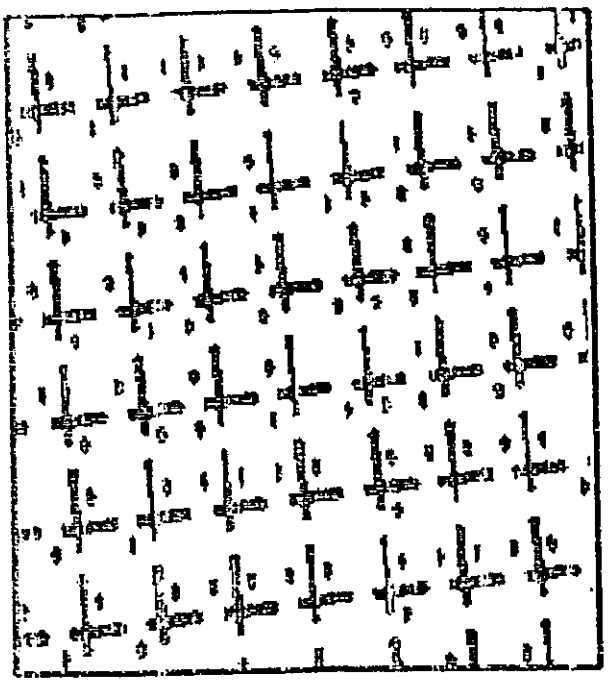


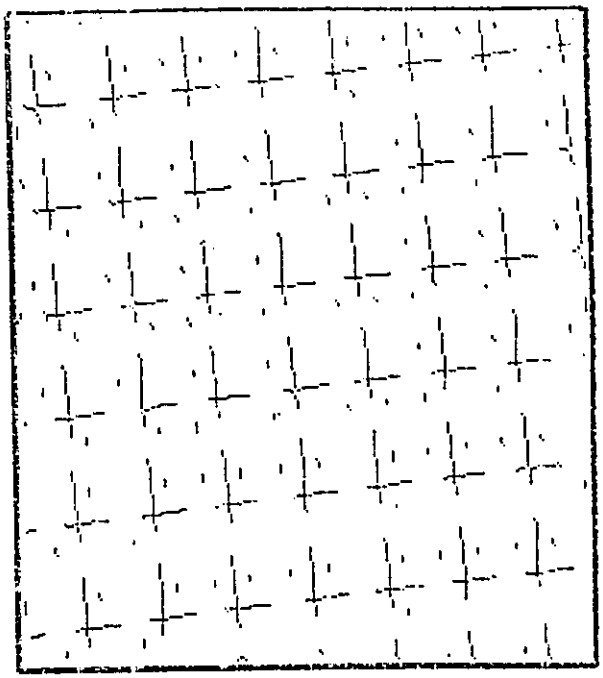


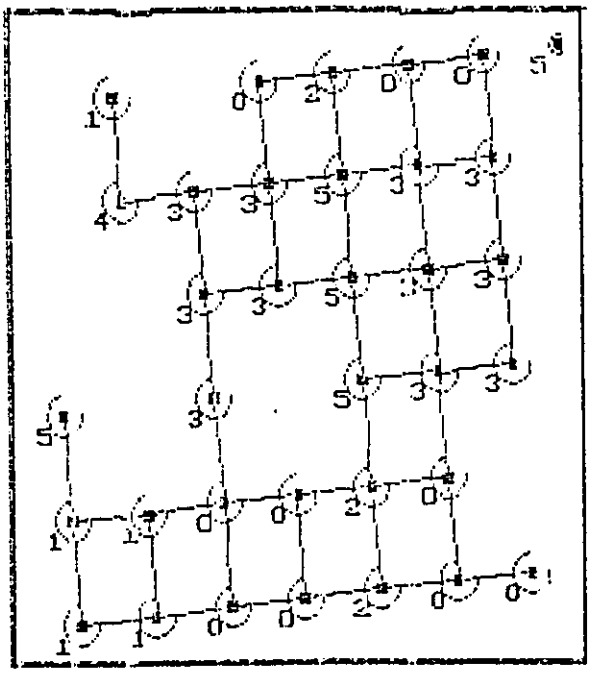
TRIANGULAR PYRAMID

Face 1

Figure 6.2.The output of the object recognition program (triangular pyramid).







TRIANGULAR PYRAMID

Face 3

Figure 6.3.The output of the object recognition program (triangular pyramid).

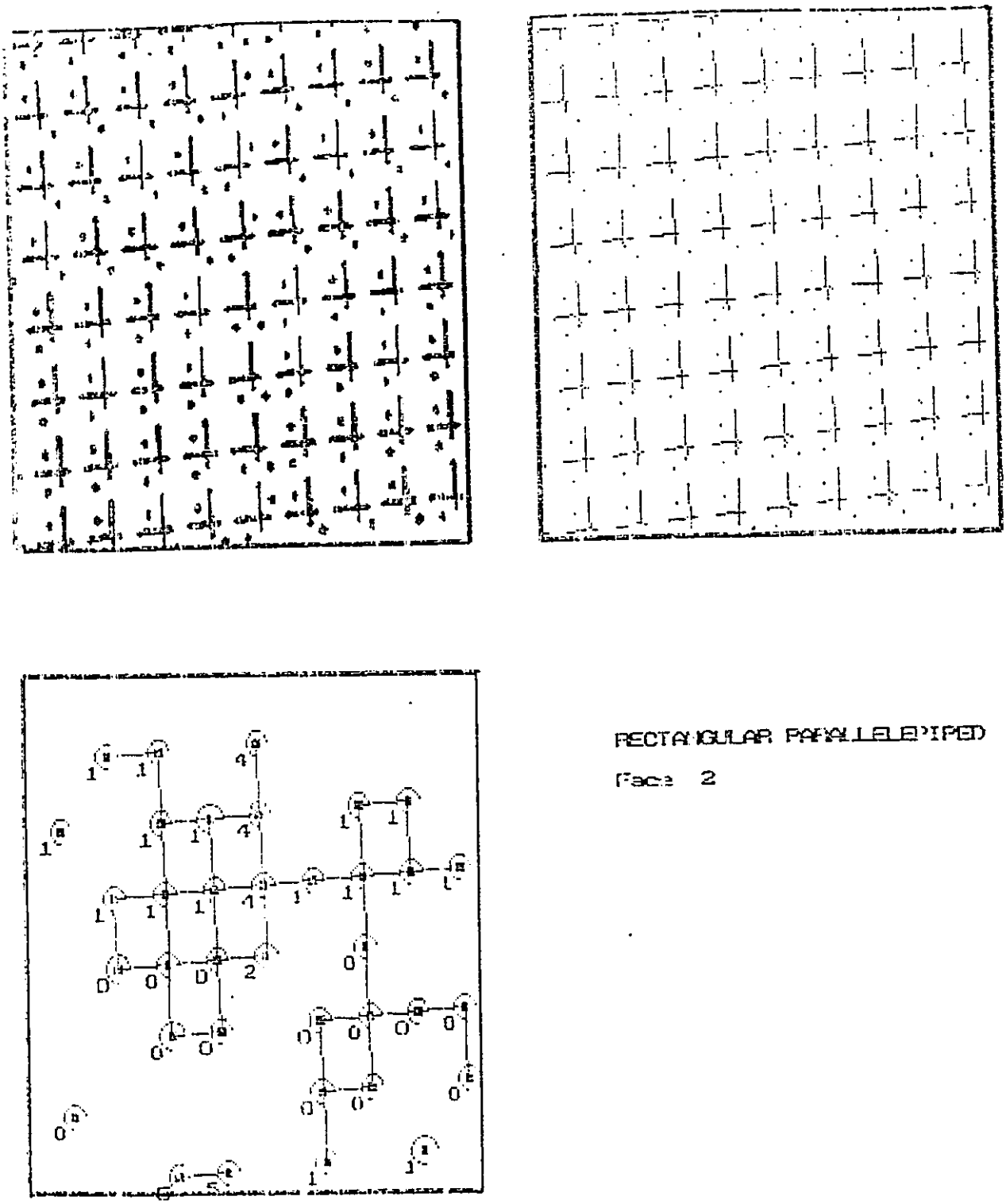


Figure 6.4.The output of the object recognition program (rectangular parallelepiped).

The POSE recovery for the rectangular parallelepiped presented in Figure 6.1 and 6.4.

Enter coordinates for point P (the image points)

```
-0.0023,  -0.0012
 0.0024,   0.0013
-0.0025,   0.0014
 0.0026,  -0.0015
```

Enter coordinates for point Q (the object points)

```
 .094,   0.073
 .013,   0.034
 .092,   0.035
 .011,   0.076
```

```
T_T:
-0.059817  0.001619 -0.315121
 0.002077 -0.068356 -0.000063
 0.003170  0.003637  1.030254

T:
-0.059817  0.002077 0.003170
 0.001619 -0.068356 0.003637
-0.315121 -0.000063 1.030254

T_1:
-17.016951      - 0.517078      0.054182
- 0.679921      -14.650069      0.053810
- 5.204974      - 0.159055      0.987210
```

```
T_T_1:
-17.016951      - 0.679921      -5.204974
- 0.517078      -14.650069      -0.159055
  0.054182        0.053810       0.987210

VLI[0]=-5.204974  VLI[1]=-0.159055  VLI[2]=0.987210

DI=0.189578 m.

theta=0.306518 rad.

VLO[0]=-0.315121     VLO[1]=-0.000063     VLO[2]=1.030254
PPO[0]= 0.054182     PPO[1]= 0.053810     PPO[2]=0.987210

DO=3.214495 m.
niu=3.141392 rad.

XSGN=1  YSGN=1

     DCP=0.969943 m.
     XCP=0.347554 m.
     YCP=0.053181 m.
     ZCP=0.924734 m.
```

PART 7

CONCLUSION

This part of the thesis presents what has been achieved and suggests future work which could be done for further development of a reliable object-recognition system. The new grid node indexing method based on PRA encoding has the notable advantage of using only one symbol per grid step, independent of the desired grid resolution. Because it uses an object model database, this object recognition technique can be easily integrated with other sensing capabilities (e.g. tactile sensing, infrared sensing) in a multi-sensor environment, perception system.

7.1. What Has Been Achieved

- All components of a model-based object-recognition system (see Figure 1.1) were implemented. The framework of this system supports all the main requirements of such a system.

- *The description process* implemented as a PRA visual symbol recognition and array reconstruction, proved to be a powerful and efficient way for permanent surface encoding.

- The relational *Object Database* that was designed, responds well to the main queries, and makes the process of storing and deleting objects from the system easy.

- *The recognition process* that was implemented as a high level query to the *Object Database*, using some low level queries, is in fact the hidden mechanism that performs the matching between the object model, stored in the database, and the (i,j) coordinates of the recognized pseudo-random window $W(i,j)$. In fact the main contribution of this thesis is to reduce the object recognition process to be reduced to a pseudo-random window recovery.

- *The rendering process* communicates to the user the found object each time an experiment is done.

7.2. Limitations

- This method is restricted to applications which allow for permanent object encoding and a priori mapping of all encoded object surfaces in a database (space station, nuclear plants, hazardous but controllable environments).

- For making the process of object modelling and POSE recovery easier only the recognition of polyhedral objects was considered.

- More robust image processing techniques can be used for a more robust symbol recognition.

5.3. Future Work

The discussed object recognition method using PRA encoding can be developed by:

- Studying methods to find the best way for the approximation of the physical objects with polyhedral objects.

- A complete integration of all the object-recognition system components (the POSE recovery is not integrated to the system at this moment).

- The rendering process can be further developed and interfaced with a robot arm for the purpose of object manipulation.

- Extension to grid-encoded light applications, where no permanent object marking would be required.

BIBLIOGRAPHY

[1] Ernest L. Hall, James B. K. Tio, Charles A. McPherson, Firooz A.Sadjad- "Measuring Curved Surfaces for Robot Vision," *IEEE Computers*, pp 42-54 Dec. 1992.

[2] Fatih Ulupinar, Ramakant Nevatia, "Perception of 3-D Surfaces from 2-D Contours," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 15, No. 1, pp. 3-18, January 1993.

[3] H.G. Barrow and J.M. Tenenbaum, "Interpreting line drawing as three dimensional surfaces," *Artif. Intell.*, vol. 17, pp. 75-116, 1981.

[4] I. Beiderman, "Recognition by components: A theory of human image understanding," *Psychology. Rev.*, vol. 94, no. 2, pp. 115-147, 1987.

[5] I. Weiss, "3-D shape representation by contours," *Computer Vision Graphics Image Processing*, vol 41, pp 80-100, 1988.

[6] K. A. Stevens, "The visual interpretation of surface contours," *Artif. Intell.*, vol 17, pp. 47-73, 1983.

[7] Paul J. Besl, Ramesh C. Jain, "Three-Dimensional Object Recognition," *Computing Surveys*, Vol. 17, No. 1 pp. 75-145, March 1985.

[8] Will P. M., Pennington K. S., "Grid Coding: A Novel Technique for Image Processing, *Proc. IEEE*, vol 60, No. 6, pp. 669-680, June 1972.

[9] Wang Y. F., Mitiche A., Aggarwal J.K., "Computation of Surface Orientation and Structure of Objects Using Grid Coding," *IEEE Tran. Pattern Anal. Machine Intell*, vol. PAMI-9 No. 1, pp. 129-137, Jan 1987.

[10] Posdemer J. L., Altschuler M. D., "Surface Measurement by Space-encoded Projected Beam Systems," *Comput. Graphics Image Processing*, vol. 18, pp. 1-17, 1982.

[11] Boyer K.L., Kak A.C., "Color-Encoded Structured Light for Rapid Active Ranging," *IEEE Trans. Pattern Anal. machine Intell*, vol. PAMI-9, No. 1, pp. 14-28, Jan. 1987.

[12] Le Moigne J. J., Waxman A. M., "Structured Light Patterns for Robot Mobility," *IEEE J. Robotics Automat*, vol. 4 No. 5, pp. 541-548, Oct. 1988.

[13] Hu G., Stockman G., "3-D Surface Solution Using Structured Light and Constraint Propagation", *IEEE Trans. Pattern Anal. Machine Intell*, Vol 11, No. 4, pp. 390-402, April 1989.

[14] Vuylsteke P., Oosterlink A., "Range Image Acquisition with a Single Binary-Encoded Light Pattern", *IEEE Trans. Pattern Anal. Machine Intell.* Vol. 12, No. 2, pp. 148-164, Feb. 1990.

[15] Fischler M. A., Bolles R. C., "Random Sample Consensus: A paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography," *Comm. ACM*, Vol. 24, No. 6, pp. 381-395, June 1981.

[16] Linnainmaa S., Harwood D., Davis L.S., "Pose Determination of a Three-Dimensional Object Using Triangle Pairs," *IEEE Trans. Pattern Anal. Machine Intell,* Vol. 10, No. 5, pp. 634-647, Sept. 1988.

[17] David C. Buchthal, Douglas E. Cameron-"Modern Abstract Algebra", *PWS Publshers*, 1987.

[18] Shu Lin, Daniel J. Costello-"Error Control Coding, Fundamentals and Applications", Prentice-Hall, Inc., *Englewood Cliffs*, N.J. 1983.

[19] F. Jessie Mac Williams and Neil J. A. Sloane, "Pseudo-random Sequences and Arrays", *Proceedings of the IEEE*, Vol. 64, No. 12 December 1976.

[20] D. H. Green, I. S. Taylor, "Ireducible polynomials over composite Galois fields and their applications in coding techniques," *Proc Inst. Elec.Eng.*, vol. 121, pp. 935-939, 1974.

[21] Harver L. Garner, "The residual Number System," *IRE Transactions on Electronic Computers*, pp. 140-147, June 1959.

[22] Robert J. McEliece ,"Finite Fields for Computer Scientists and Engineers," *Kluwer Academic Publishers* 1987.

[23] C. Balza, A. Fromageot, and M. Maniere, "Four-level pseudo-random sequences," *Electron. Lett.*, vol. 3, pp 313-315, 1967.

[24] P.A.N. Briggs and K.R. Godfrey, "Autocorrelation function of 4-level m-sequences," *Electron. Lett.*, vol. 4, pp. 232-233, 1963.

[25] S.W. Golumb, Ed., Digital Communications with Space Applications, *Englewood Cliffs*, N.J., Prince-Hall, 1966.

[26] E. M. Petriu "Absolute-type position transducers using a pseudo-random encoding," *IEEE Tran. Instrum. Meas.*, vol. IM-36, no. 4, pp. 950-955, Feb. 1987.

[27] E.Petriu., Basran J. S., Groen F. C. A., "Automated Guided Vehicle Position Recovery," *IEEE Tran. Instrum. Meas.*, vol. 39, no.1, pp 254-258, Feb. 1990.

[28] R. Spann, "A Two-Dimensional correlation property of pseudo-random maximal-length sequences," *Proc. IEEE*, vol 53, p2137, 1963.

[29] Ming Xie, "Edge linking by using causal neighborhood window", *Pattern Recognition Letters 13* Sept. 1992, North-Holland pp. 649.

[30] R. M. Haralick and L.G. Shapiro, "Computer and Robot Vision," vol. 1, *Addison-Wesley Publ. Co.*, Reading, MA, 1992.

[31] J. Hilditch, "Linear Skeleton from Square Cupboards," *Machine Intelligence*, Vol. 4, (B. Meltzer and D. Michie, Eds.), Edinburgh University Press, Edinburgh, pp. 403-420, 1969.

[32] E. Petriu, W.S. McMath, S.K. Yeung, N. Trif, T. Bieseman "Two-Dimensional Position Recovery for a Free-Ranging Automated Guided Vehicle," IEEE *Tran. Instrum. Meas.*, June 1993, (accepted).

[33] N. Trif, E.M. Petriu, W.S. McMath, S.K. Yeung "2-D Pseudo-Random Encoding for Vision-Based Vehicle Guidance," *IEEE/SAE Intelligent Vehicle Symposium '93*, July 1993,Tokyo (accepted).

[34] A. Stevens, "C Database Development," Portland, Oregon, 1991.

[35] R.J. Holt and A.N.Netravali, "Camera Calibration Problem: Some New Results," *CVGIP: Image Understanding,* Vol. 54, No. 3, pp-368-383, Nov. 1991.

[36] Rogers, D.P. and Adams, J.A. "Mathematical Elements for Computer Graphics", *McGraw Hill*, New York 1976.

[37] E. Petriu, N. Trif, W.S. McMath, S.K. Yeung, "Automated Guided Vehicle Position Recovery Using Pseudo-Random Grid Encoding," *Proc. Int. Conf. Intell. Autonomous Syst. IAS-3*, pp. 104-113, Pittsburgh, PA, 1993.

[38] A. M. Meystel, "Intelligent Motion Control in Antropomorphic Machines," in *Application in Artificial Intelligence*, (Edit S.J. Andriole), pp. 317-351, Petrocelli Books Inc., Princeton, 1985.

APPENDIX

APPENDIX A1

Primitive Polynomials over GF(q), [19]

m	GF(2)	GF(3)	$GF(2^2)$	$GF(2^3)$
1	$x+1$	$x+1$	$x+\omega$	$x+\alpha$
2	x^2+x++1	x^2+x+2	$x^2+x+\omega$	$x^2+\alpha x+\alpha$
3	x^3+x+1	x^3+2x+1	$x^3+x^2+x+\omega$	$x^3+x+\alpha$
4	x^4+x+1	x^4+x+2	$x^4+x^2+\omega x+\omega^2$	$x^4+x+\alpha^3$
5	x^5+x^2+1	x^5+2x+1	$x^5+x+\omega$	$x^5+x^2+x+\alpha^3$
6	x^6+x+1	x^6+x+2	$x^6+x^2+x+\omega$	$x^6+x+\alpha$
7	x^7+x+1	$x^7+x^6+x^4+1$	$x^7+x^2+\omega x+\omega^2$	$x^7+x^2+\alpha x+\alpha^3$
8	$x^8+x^6+x^5+x+1$	x^8+x^5+2	$x^8+x^3+x+\omega$	
9	x^9+x^4+1	$x^9+x^7+x^5+1$	$x^9+x^2+x+\omega$	
10	$x^{10}+x^3+1$	$x^{10}+x^9+x^7+2$	$x^{10}+x^3+\omega(x^2+x+1)$	
11	$x^{11}+x^2+1$			
12	$x^{12}+x^7+x^4+x^3+x+1$			
13	$x^{13}+x^4+x^3+x+1$			
14	$x^{14}+x^{12}+x^{11}+x+1$			
15	$x^{15}+x+1$			

where

ω- is a generating element for $GF(2^2)$

α-is a generating element for $GF(2^3)$.

APPENDIX A2.

The Elements of Some Galois Fields [20], [22]

0	∞
1	0

Table A2.1. The elements of GF(2).

00	$-\infty$
10	0
01	1
11	2

Table A2.2. The elements of $GF(2^2)$ generated by $h(x)=x^2+x+1$.

000	$-\infty$
100	0
010	1
001	2
110	3
011	4
111	5
101	6

Table A2.3. The elements of $GF(2^3)$ generated by $h(x)=x^3+x+1$.

0000	$-\infty$
1000	0
0100	1
0010	2
0001	3
1100	4
0110	5
0011	6
1101	7
1010	8
0101	9
1110	10
0111	11
1111	12
1011	13
1001	14

Table A2.4. The elements of the GF(2^4) generated by $h(x)=x^4+x+1$.

00000	-∞	11111	15
10000	0	11011	16
01000	1	11001	17
00100	2	11000	18
00010	3	01100	19
00001	4	00110	20
10100	5	00011	21
01010	6	10101	22
00101	7	11110	23
10110	8	01111	24
01011	9	10011	25
10001	10	11101	26
11100	11	11010	27
01110	12	01101	28
00111	13	10010	29
10111	14	01001	30

Table A2.5. The elements of GF(2^5) generated by $h(x)=x^5+x^2+1$.

000000	-∞	001111	20	010111	42
100000	0	110111	21	111011	43
010000	1	101011	22	101101	44
001000	2	100101	23	100110	45
000100	3	100010	24	010011	46
000010	4	010001	25	111001	47
000001	5	111000	26	101100	48
110000	6	011100	27	010110	49
011000	7	001110	28	001011	50
001100	8	000111	29	110101	51
000110	9	110011	30	101010	52
000011	10	101001	31	010101	53
110001	11	100100	32	111010	54
101000	12	010010	33	011101	55
010100	13	001001	34	111110	56
001010	14	110100	35	011111	57
000101	15	011010	36	111111	58
110010	16	001101	37	101111	59
011001	17	110110	38	100111	60
111100	18	011011	39	100011	61
011110	19	111101	40	100001	62
		101110	41		

Table A2.6. The elements of GF(2^6) generated by $h(x)=x^6+x+1$.

0	$-\infty$
1	0
2	1

Table A2.7. The elements of GF(3).

00	$-\infty$
10	0
01	1
12	2
22	3
20	4
02	5
21	6
11	7

Table A2.8. The elements of GF(3^2) generated by $h(x)=x^2+x+1$.

APPENDIX A3.

Data Definition Language for Object Database

OBJECT.h

```
#define APPLICATION_H

typedef enum elements {
        OBJECT_ID=1,
        OBJECT_NAME,
        NUMBER_OF_FACES,
        NUMBER_OF_VERTICES,
        VERTEX_ID,
        X_COORD,
        Y_COORD,
        Z_COORD,
        COORD_ID,
        X_POS,
        Y_POS,
        X_REL,
        Y_REL,
        FACE_ID,
        FACE_NAME,
        NR_OF_EDGES,
        FACE_SURFACE,
        GRID_STEP,
        EDGE_ID,
        LENGTH_OF_EDGE,
        TermElement = 32367
} ELEMENT;

typedef enum files {
        OBJECTS,
        VERTICES,
        PLANAR,
        FACES,
        EDGES,
        F_TO_V,
        E_TO_V,
        TermFile = 32367
} DBFILE;

struct objects {
        char object_id [4];
        char object_name [26];
        char number_of_vertices [3];
        char number_of_faces [3];
};
```

```
struct vertices {
        char vertex_id [3];
        char object_id [4];
        char x_coord [8];
        char y_coord [8];
        char z_coord [8];
};

struct planar {
        char coord_id [3];
        char vertex_id [3];
        char x_pos [8];
        char y_pos [8];
        char x_rel [8];
        char y_rel [8];
};

struct faces {
        char face_id [3];
        char object_id [4];
        char face_name [21];
        char nr_of_edges [3];
};

struct edges {
        char edge_id [3];
        char face_id [3];
        char length_of_edge [8];
};

struct f_to_v {
        char vertex_id [3];
        char face_id [3];
};


struct e_to_v {
        char vertex_id [3];
        char edge_id [3];
};

#include "cdata.h"
```

OBJECT.c

```
#include "OBJECT.h"

const char *denames [] = {
        "OBJECT_ID",
        "OBJECT_NAME",
        "NUMBER_OF_FACES",
        "NUMBER_OF_VERTICES",
        "VERTEX_ID",
        "X_COORD",
        "Y_COORD",
        "Z_COORD",
        "COORD_ID",
        "X_POS",
        "Y_POS",
        "X_REL",
        "Y_REL",
        "FACE_ID",
        "FACE_NAME",
        "NR_OF_EDGES",
        "FACE_SURFACE",
        "GRID_STEP",
        "EDGE_ID",
        "LENGTH_OF_EDGE",
        NULL
};

const char eltype [] = "AAAAAAAAAAAAAAAAAAAA";

const char *elmask [] = {
        "    ",
        "____",
        "                              ",
        "______________________________",
        "    ",
        "____",
        "    ",
        "____",
        "    ",
        "____",
        "            ",
        "____________",
        "            ",
        "____________",
        "            ",
        "____________",
        "    ",
        "____",
        "            ",
        "____________",
        "            ",
        "____________",
        "            ",
        "____________",
        "            ",
        "____________",
        "    ",
        "____",
        "                                ",
        "________________________________",
        "    ",
        "____",
        "            ",
        "____________",
        "            ",
        "____________",
        "    ",
        "____",
        "            "
        "____________"
```

```
};

const char *dbfiles [] = {
        "OBJECTS",
        "VERTICES",
        "PLANAR",
        "FACES",
        "EDGES",
        "F_TO_V",
        "E_TO_V",
        NULL
};


const int ellen [] = {
        3,25,2,2,2,7,7,7,2,7,7,7,7,2,20,2,7,7,2,7
};

const ELEMENT f_objects [] = {
        OBJECT_ID,
        OBJECT_NAME,
        NUMBER_OF_VERTICES,
        NUMBER_OF_FACES,
        0
};

const ELEMENT f_vertices [] = {
        VERTEX_ID,
        OBJECT_ID,
        X_COORD,
        Y_COORD,
        Z_COORD,
        0
};

const ELEMENT f_planar [] = {
        COORD_ID,
        VERTEX_ID,
        X_POS,
        Y_POS,
        X_REL,
        Y_REL,
        0
};

const ELEMENT f_faces [] = {
        FACE_ID,
        OBJECT_ID,
        FACE_NAME,
        NR_OF_EDGES,
        0
};
```

```
const ELEMENT f_edges [] = {
        EDGE_ID,
        FACE_ID,
        LENGTH_OF_EDGE,
        0
};

const ELEMENT f_f_to_v [] = {
        VERTEX_ID,
        FACE_ID,
        0
};

const ELEMENT f_e_to_v [] = {
        VERTEX_ID,
        EDGE_ID,
        0
};

const ELEMENT *file_ele [] = {
        f_objects,
        f_vertices,
        f_planar,
        f_faces,
        f_edges,
        f_f_to_v,
        f_e_to_v,
        0
};

const ELEMENT x1_objects [] = {
        OBJECT_ID,
        0
};

const ELEMENT *x_objects [] = {
        x1_objects,
        NULL
};

const ELEMENT x1_vertices [] = {
        VERTEX_ID,
        OBJECT_ID,
        0
};

const ELEMENT x2_vertices [] = {
        VERTEX_ID,
        0
};
```

```
const ELEMENT x3_vertices [] = {
        OBJECT_ID,
        0
};

const ELEMENT *x_vertices [] = {
        x1_vertices,
        x2_vertices,
        x3_vertices,
        NULL
};

const ELEMENT x1_planar [] = {
        COORD_ID,
        VERTEX_ID,
        0
};

const ELEMENT x2_planar [] = {
        COORD_ID,
        0
};

const ELEMENT x3_planar [] = {
        VERTEX_ID,
        0
};

const ELEMENT *x_planar [] = {
        x1_planar,
        x2_planar,
        x3_planar,
        NULL
};

const ELEMENT x1_faces [] = {
        FACE_ID,
        OBJECT_ID,
        0
};

const ELEMENT x2_faces [] = {
        FACE_ID,
        0
};

const ELEMENT x3_faces [] = {
        OBJECT_ID,
        0
};
```

```
const ELEMENT *x_faces [] = {
        x1_faces,
        x2_faces,
        x3_faces,
        NULL
};

const ELEMENT x1_edges [] = {
        EDGE_ID,
        FACE_ID,
        0
};

const ELEMENT x2_edges [] = {
        EDGE_ID,
        0
};

const ELEMENT x3_edges [] = {
        FACE_ID,
        0
};

const ELEMENT *x_edges [] = {
        x1_edges,
        x2_edges,
        x3_edges,
        NULL
};

const ELEMENT x1_f_to_v [] = {
        VERTEX_ID,
        FACE_ID,
        0
};

const ELEMENT x2_f_to_v [] = {
        VERTEX_ID,
        0
};

const ELEMENT x3_f_to_v [] = {
        FACE_ID,
        0
};

const ELEMENT *x_f_to_v [] = {
        x1_f_to_v,
        x2_f_to_v,
        x3_f_to_v,
        NULL
};
```

```
const ELEMENT x1_e_to_v [] = {
        VERTEX_ID,
        EDGE_ID,
        0
};

const ELEMENT x2_e_to_v [] = {
        VERTEX_ID,
        0
};

const ELEMENT x3_e_to_v [] = {
        EDGE_ID,
        0
};

const ELEMENT *x_e_to_v [] = {
        x1_e_to_v,
        x2_e_to_v,
        x3_e_to_v,
        NULL
};

const ELEMENT **index_ele [] = {
        x_objects,
        x_vertices,
        x_planar,
        x_faces,
        x_edges,
        x_f_to_v,
        x_e_to_v,
        NULL
};


#ifdef NULL_IS_DEFINED
        #undef NULL
        #undef NULL_IS_DEFINED
#endif
```

APPENDIX A4.

Source Code for POSE Recovery

```
// ------------------------------------------------------------------
// PROBLEM STATEMENT:
// GIVEN: -a coorespondence between four points lying in a plane in 3-D space
//        (called the object plane), and four points lying in a distinct
//        plane (called the image plane);
//        -the distance between the center of perspective and the image plane
//        (the focal length f of the image system);
//        -the principal point of the image plane ( the location, in image
//        coordinates, of the point at which the optical axis of the length
//        pierces the image plane.
//
//FIND:  -the location of the center of perspective relative to the
//       coordinate system of the object plane.
// ------------------------------------------------------------------
// NOTATION:
//  -P[3][4] -the image point matrix;
//  -Q[3][4] -the object point matrix;
//  -PPI -the origin of the image and the principal point of the image;
//  -the object coordinate system has the equation Z=0;
//  -homogeneous coordinates will be assumed;
// ------------------------------------------------------------------
// SOLUTION
// To determine the location of the center of perspective relative to the
// coordinate system of the object plane we will follow 10 steps:
// STEP  1: determine the T[3][3] collineation matrix which maps points
//          from the object plane to image plane;
//
// STEP  2: determine the vanishing line in the image plane VLI[3];
//
// STEP  3: determine the distance DI from the origin of the image plane (PPI)
//          to the vanishing line VLI;
//
// STEP  4: determine the dihedral (tilt) angle theta between the image and
//          object planes ( the focal length f has to be known);
//
// STEP  5: determine the vanishing line in the object plane VLO;
//
// STEP  6: determine the location of PPO in the object plane. The PPO is the
//          point at which the optical axis of the lense pierces the object
//          plane;
//
// STEP  7: determine the distance DO from PPO to the vanising line VLO in
//          the object plane;
//
```

```
// STEP  8: determine the "pan" angle dollar as the angle between the normal
//          to VLO and the X axis in the object plane;
//
// STEP  9: determine XSGN and YSGN. If a line (parallel to the X axis in the
//          object plane) through PPO intersects VLO to the right of PPO,
//          then XSGN=1. Otherwise XSGN=-1. Simmilarly for YSGN.
//
//STEP 10: determine the location of CP( XCP, YCP, ZCP in the object
//          plane coordinate system;
// ------------------------------------------------------------------------
// This method was taken from Martin A. Fischler and Robert C. Bolles
// SRI International "Random Sample Consensus: A Paradigm for Model
// Fitting with Application to Image Analysis and Automated Cartography",
// Graphical and Image Processing June 1981 Vol. 24 Nr. 6
// Written by Niculaie Trif
// Feb.11.1993
// ------------------------------------------------------------------------
// header files

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

// ------------------------------------------------------------------------
// functions

void input(float *, float *);
void calc_mat_3(float *, float *);
float *alloc_memory(int);
float calc_det(float *);
void minor_a(int i, int j, float *, float, float *);
float calc_det_2(void);
void calc_inv_A(float *, float, float *);
void step_1(void);
void step_2(void);
void step_3(void);
void step_4(void);
void step_5(void);
void step_6(void);
void step_7(void);
void step_8(void);
void step_9(void);
void step_10(void);

// ------------------------------------------------------------------------
// variables declaration

double P[4][3];
double Q[4][3]
float temp[2][2];
double theta, f=0.06 ;    // in the futere should be obtained directly from the camera
```

```
int XSGN, YSGN;
float *float_nr_P, *float_nr_Q;
float *float_P_3, *float_Q_3;
float *inv_P, *inv_Q, *inv_T;
float *float_T;
float det_P, det_Q, det_T;
float V[3], R[3], W[3][3], t[3][3], T_T[3][3], P_3[3][3],
     T[3][3], T_1[3][3], T_T_1[3][3], VLI[3], VLO[3], PPO[3];
double DI,
       DO,
       dollar,
       DCP,
       XCP,
       YCP,
       ZCP,
       sincos,
       sinsin;
// ------------------------------------------------------------------------

int main(void)
{
// alloc memory
     float_nr_P=alloc_memory(12);
     float_nr_Q=alloc_memory(12);

// Get input data
     clrscr();
     input(float_nr_P, float_nr_Q);
     clrscr();

// The ten steps required to solve the problem
     step_1();
     step_2();
     step_3();
     step_4();
     step_5();
     step_6();
     step_7();
     step_8();
     step_9();
     step_10();

getch();
return (0);
}

void step_1(void)
{
// STEP 1.-determine the colination matrix T[3][3]
// this will require few substeps

int i,j;
```

```
printf("\nSTEP 1: The collineation matrix T.\n");
// construct matrix P and Q
    calc_mat_3(float_nr_P, float_nr_Q);

// determine inv_P and inv_Q
    det_P=calc_det(float_P_3);
    det_Q=calc_det(float_Q_3);
    inv_P=alloc_memory(9);
    inv_Q=alloc_memory(9);
    calc_inv_A(float_P_3, det_P, inv_P);
    calc_inv_A(float_Q_3, det_Q, inv_Q);

// determine V
    V[0]=(*(inv_P+0)*(*(float_nr_P+9)))+
      (*(inv_P+3)*(*(float_nr_P+10)))+
      (*(inv_P+6));
    V[1]=(*(inv_P+1)*(*(float_nr_P+9)))+
      (*(inv_P+4)*(*(float_nr_P+10)))+
      (*(inv_P+7));
    V[2]=(*(inv_P+2)*(*(float_nr_P+9)))+
      (*(inv_P+5)*(*(float_nr_P+10)))+
      (*(inv_P+8));

// determine R
    R[0]=(*(inv_Q+0)*(*(float_nr_Q+9)))+
      (*(inv_Q+3)*(*(float_nr_Q+10)))+
      (*(inv_Q+6));
    R[1]=(*(inv_Q+1)*(*(float_nr_Q+9)))+
      (*(inv_Q+4)*(*(float_nr_Q+10)))+
      (*(inv_Q+7));
    R[2]=(*(inv_Q+2)*(*(float_nr_Q+9)))+
      (*(inv_Q+5)*(*(float_nr_Q+10)))+
      (*(inv_Q+8));

// determine W[3][3]
    for(i=0;i<3;i++)
      for(j=0;j<3;j++)
      W[i][j]=0;
    W[0][0]=(V[0]*R[2])/(R[0]*V[2]);
    W[1][1]=(V[1]*R[2])/(R[1]*V[2]);
    W[2][2]=1;

// determine T transposed
   t[0][0]=(*(inv_Q+0))*W[0][0];
   t[0][1]=(*(inv_Q+1))*W[1][1];
   t[0][2]=(*(inv_Q+2));
   t[1][0]=(*(inv_Q+3))*W[0][0];
   t[1][1]=(*(inv_Q+4))*W[1][1];
   t[1][2]=(*(inv_Q+5));
   t[2][0]=(*(inv_Q+6))*W[0][0];
   t[2][1]=(*(inv_Q+7))*W[1][1];
```

```
  t[2][2]=(*(inv_Q+8));

T_T[0][0]=t[0][0]*P_3[0][0]+t[0][1]*P_3[1][0]+t[0][2]*P_3[2][0];
T_T[0][1]=t[0][0]*P_3[0][1]+t[0][1]*P_3[1][1]+t[0][2]*P_3[2][1];
T_T[0][2]=t[0][0]*P_3[0][2]+t[0][1]*P_3[1][2]+t[0][2]*P_3[2][2];
T_T[1][0]=t[1][0]*P_3[0][0]+t[1][1]*P_3[1][0]+t[1][2]*P_3[2][0];
T_T[1][1]=t[1][0]*P_3[0][1]+t[1][1]*P_3[1][1]+t[1][2]*P_3[2][1];
T_T[1][2]=t[1][0]*P_3[0][2]+t[1][1]*P_3[1][2]+t[1][2]*P_3[2][2];
T_T[2][0]=t[2][0]*P_3[0][0]+t[2][1]*P_3[1][0]+t[2][2]*P_3[2][0];
T_T[2][1]=t[2][0]*P_3[0][1]+t[2][1]*P_3[1][1]+t[2][2]*P_3[2][1];
T_T[2][2]=t[2][0]*P_3[0][2]+t[2][1]*P_3[1][2]+t[2][2]*P_3[2][2];

printf("\nT_T:\n");
for(i=0;i<3;i++)
   {
   for(j=0;j<3;j++)
      printf("%f\t",T_T[i][j] );
   printf("\n");
   }
getch();

// determine T
float_T=alloc_memory(9);
for(i=0;i<3;i++)
  for(j=0;j<3;j++)
     {
     T[i][j]=T_T[j][i];
     *(float_T+3*i+j)=T_T[j][i];
     }

printf("\nT:\n");
for(i=0;i<3;i++)
   {
   for(j=0;j<3;j++)
      printf("%f\t",T[i][j] );
   printf("\n");
   }
getch();
// determine inv_T
det_T=calc_det(float_T);
inv_T=alloc_memory(9);
calc_inv_A(float_T, det_T, inv_T);
for(i=0;i<3;i++)
   for(j=0;j<3;j++)
      T_1[i][j]=*(inv_T+3*i+j);

printf("\nT_1:\n");
for(i=0;i<3;i++)
   {
   for(j=0;j<3;j++)
      printf("%f\t",T_1[i][j] );
   printf("\n");
```

```
  }
getch();

// determine inv T transposed
for(i=0;i<3;i++)
  for(j=0;j<3;j++)
     T_T_1[i][j]=T_1[j][i];

printf("\nT_T_1:\n");
for(i=0;i<3;i++)
   {
   for(j=0;j<3;j++)
      printf("%f\t",T_T_1[i][j] );
   printf("\n");
   }
getch();
}


void step_2(void)
{
// STEP 2.
// determine the vanibing line  in the image plane VLI[3]
int i,j;
printf("\n");
for(i=0;i<3;i++)
   VLI[i]=T_T_1[i][2];
for (i=0;i<3;i++)
   printf("VLI[%d]=%f\t",i,VLI[i]);
printf("\n");
}

void step_3(void)
{
// STEP 3.
// determining the distance DI between the origin of image plane PPI
// to the vanishing line VLI[3]
printf("\n");
DI=( (VLI[2])/(sqrt(VLI[0]*VLI[0]+VLI[1]*VLI[1]) ));
if(DI < 0)
   DI=-DI;
printf("DI=%f m.\n",DI);
}

void step_4(void)
{
// STEP 4.
// determine the dihedral (tilt) angle "theta" between the image and
// object planes
printf("\n");
theta=atan2(f,DI);
printf ("theta=%f rad.\n",theta);
```

```
}
void step_5(void)
{
// STEP 5.
// determine the vanishing line in the object plane VLO[3]
int i;
printf("\n");
for(i=0;i<3;i++)
   {
   VLO[i]=T_T[i][2];
   printf("VLO[%d]=%f \t",i,VLO[i]);
   }
printf("\n");
}

void step_6(void)
{
// STEP 6.
// determine the location of point PPO in the object plane
// PPO -is the point at which the optical axis of the lense pierces the
//      object plane
int i;
for(i=0;i<3;i++)
   {
   PPO[i]=T_T_1[2][i];
   printf("PPO[%d]=%f  \t",i, PPO[i]);
   }
printf("\n");
}

void step_7(void)
{
// STEP 7.
// determine the distance DO between PPO and the vanising line
// in the object plane VLO

DO=(VLO[0]*PPO[0]+VLO[1]*PPO[1]+VLO[2]*PPO[2])/
   (PPO[2]*(sqrt(VLO[0]*VLO[0]+VLO[1]*VLO[1])));
printf("\nDO=%f m.",DO);
}

void step_8(void)
{
// STEP 8.
// determine the "pan" angle "dollar" as the angle between the normal to VLO
// and the X axis
dollar=atan2(-VLO[1], VLO[0]);
printf("\n$=%f",dollar);
}

void step_9(void)
{
```

```
// STEP 9.
// determine XSGN and YSGN; if a line (parallel to the X axis in the object
// plane) through PPO intersect VLO to the right of PPO then XSGN=1
// otherwise XSGN=-1. Simmilarly for YSGN
if(( (VLO[0]*PPO[0]+VLO[1]*PPO[1]+VLO[2]*PPO[2])/(VLO[0]*PPO[2]) ) <0 )
  XSGN=1;
else
 XSGN=-1;

if(( (VLO[0]*PPO[0]+VLO[1]*PPO[1]+VLO[2]*PPO[2])/(VLO[1]*PPO[2]) ) <0 )
  YSGN=1;
else
 YSGN=-1;
printf("\nXSGN=%d\tYSGN=%d\n",XSGN, YSGN);
}

void step_10(void)
{
// STEP 10.
//determine the location of the CP in the object plane coordinate system
DCP=DO*sin(theta);
if( (sincos=DCP*sin(theta)*cos(dollar) ) >0 )
  XCP=XSGN*sincos+PPO[0]/PPO[2];
else
  XCP=XSGN*(-1)*sincos+PPO[0]/PPO[2];

if( (sinsin=DCP*sin(theta)*sin(dollar) ) >0 )
  YCP=YSGN*sinsin+PPO[1]*PPO[2];
else
  YCP=YSGN*(-1)*sinsin+PPO[1]/PPO[2];

ZCP=DCP*cos(theta);

printf("\n\tDCP=%f\n\tXCP=%f\n\tYCP=%f\n\tZCP=%f\n",DCP,XCP, YCP, ZCP);
}

void calc_inv_A(float *float_nr, float det_mat_a, float *result)
{
int i,j;

for(i=0;i<3;i++)
  for(j=0;j<3;j++)
    {
    minor_a(i+1,j+1,float_nr, det_mat_a, result);
    }
}

float calc_det(float *ptr_float)
{
return( ( *(ptr_float+0) * (*(ptr_float+4)) * (*(ptr_float+8)) ) +
    ( *(ptr_float+1) * (*(ptr_float+5)) * (*(ptr_float+6)) ) +
    ( *(ptr_float+2) * (*(ptr_float+7)) * (*(ptr_float+3)) ) -
```

```
    ( *(ptr_float+2) * (*(ptr_float+4)) * (*(ptr_float+6)) ) -
    ( *(ptr_float+5) * (*(ptr_float+7)) * (*(ptr_float+0)) ) -
    ( *(ptr_float+8) * (*(ptr_float+3)) * (*(ptr_float+1)) ) );
}

void calc_mat_3(float *ptr_float_1, float *ptr_float_2)
{
int i,j;

float_P_3=alloc_memory(9);
float_Q_3=alloc_memory(9);

for(i=0;i<9;i++)
   {
   *(float_P_3+i)=*(ptr_float_1+i);
   *(float_Q_3+i)=*(ptr_float_2+i);
   }
for(i=0;i<3;i++)
  for(j=0;j<3;j++)
     P_3[i][j]=*(float_P_3+3*i+j);
}


void input(float *ptr_float_1, float *ptr_float_2)
{
int i,j;

clrscr();
printf("\nEnter coordinates for point P (the image points).\n");
for(i=0;i<4;i++)
   for(j=0;j<3;j++)
      {
      *(ptr_float_1+3*i+j)=P[i][j];
      scanf("%f",(ptr_float_1+3*i+j) );
      }

clrscr();
printf("\nEnter coordinates for point Q (the object points).\n");
for(i=0;i<4;i++)
   for(j=0;j<3;j++)
      {
      *(ptr_float_2+3*i+j)=Q[i][j];
      scanf("%f",(ptr_float_2+3*i+j) );
      }
}


float calc_det_2()
   {
   return (temp[0][0]*temp[1][1]-temp[0][1]*temp[1][0]);
   }
```

```
float *alloc_memory(int nr_elem)
{
float *temp;
temp=(float *)malloc(nr_elem*(sizeof(float)));
if(temp==NULL)
  {
  printf("Fail to alloc memory!!!");
  exit(1);
  }
return (temp);

}

void minor_a(int i, int j, float *float_nr, float det_mat_a, float *minor)
{
//printf("det_mat_a=%f",det_mat_a);
//getch();
switch (i)
     {
     case 1:
     switch (j)
        {
        case 1:
        temp[0][0]=*(float_nr+4);
        temp[0][1]=*(float_nr+5);
        temp[1][0]=*(float_nr+7);
        temp[1][1]=*(float_nr+8);
        *(minor+0)=calc_det_2()/det_mat_a;
        break;

        case 2:
        temp[0][0]=*(float_nr+3);
        temp[0][1]=*(float_nr+5);
        temp[1][0]=*(float_nr+6);
        temp[1][1]=*(float_nr+8);
        *(minor+3)=-calc_det_2()/det_mat_a;
        break;

        case 3:
        temp[0][0]=*(float_nr+3);
        temp[0][1]=*(float_nr+4);
        temp[1][0]=*(float_nr+6);
        temp[1][1]=*(float_nr+7);
        *(minor+6)=calc_det_2()/det_mat_a;
        break;
        }
     break;
     case 2:
     switch (j)
        {
        case 1:
        temp[0][0]=*(float_nr+1);
```

```
    temp[0][1]=*(float_nr+2);
    temp[1][0]=*(float_nr+7);
    temp[1][1]=*(float_nr+8);
    *(minor+1)=-calc_det_2()/det_mat_a;
    break;

    case 2:
    temp[0][0]=*(float_nr+0);
    temp[0][1]=*(float_nr+2);
    temp[1][0]=*(float_nr+6);
    temp[1][1]=*(float_nr+8);
    *(minor+4)=calc_det_2()/det_mat_a;
    break;

    case 3:
    temp[0][0]=*(float_nr+0);
    temp[0][1]=*(float_nr+1);
    temp[1][0]=*(float_nr+6);
    temp[1][1]=*(float_nr+7);
    *(minor+7)=-calc_det_2()/det_mat_a;
    break;
    }
break;

case 3:
switch (j)
    {
    case 1:
    temp[0][0]=*(float_nr+1);
    temp[0][1]=*(float_nr+2);
    temp[1][0]=*(float_nr+4);
    temp[1][1]=*(float_nr+5);
    *(minor+2)=calc_det_2()/det_mat_a;
    break;

    case 2:
    temp[0][0]=*(float_nr+0);
    temp[0][1]=*(float_nr+2);
    temp[1][0]=*(float_nr+3);
    temp[1][1]=*(float_nr+5);
    *(minor+5)=-calc_det_2()/det_mat_a;
    break;

    case 3:
    temp[0][0]=*(float_nr+0);
    temp[0][1]=*(float_nr+1);
    temp[1][0]=*(float_nr+3);
    temp[1][1]=*(float_nr+4);
    *(minor+8)=calc_det_2()/det_mat_a;
    break;
    }
 break;
```