Two new parallel processors for real time classification of 3-D moving objects and quad tree generation

Farjam Majd
Portland State University

AN ABSTRACT OF THE THESIS OF Farjam Majd for the Master of Science in
Engineering: Electrical and Computer presented November 14, 1985.

Title: Two New Parallel Processors for Real Time Classification of 3-D
        Moving Objects and Quad Tree Generation

APPROVED BY MEMBERS OF THE THESIS COMMITTEE:

Dr. F Badi'i, Chairman

Dr. J. Heneghan

Prof. J. Riley

Dr. M. Ghafarzadeh

        Two related image processing problems are addressed in this thesis.
First, the problem of identification of 3-D objects in real time is
explored. An algorithm to solve this problem and a hardware system for
parallel implementation of this algorithm are proposed. The
classification scheme is based on the "Invariant Numerical Shape
Modeling" (INSM) algorithm originally developed for 2-D pattern
recognition such as alphanumeric characters. This algorithm is then
extended to 3-D and is used for general 3-D object identification. The
hardware system is an SIMD parallel processor, designed in bit slice

fashion for expandability. It consists of a library of images coded according to the 3-D INSM algorithm and the SIMD classifier which compares the code of the unknown image to the library codes in a single clock pulse to establish its identity. The output of this system consists of three signals: U, for unique identification; M, for multiple identification; and N, for non-identification of the object.

Second, the problem of real time image compaction is addressed. The quad tree data structure is described. Based on this structure, a parallel processor with a tree architecture is developed which is independent of the data entry process, i.e., data may be entered pixel by pixel or all at once. The hardware consists of a tree processor containing a tree generator and three separate memory arrays, a data transfer processor, and a main memory unit. The tree generator generates the quad tree of the input image in tabular form, using the memory arrays in the tree processor for storage of the table. This table can hold one picture frame at a given time. Hence, for processing multiple picture frames the data transfer processor is used to transfer their respective quad trees from the tree processor memory to the main memory. An algorithm is developed to facilitate the determination of the connections in the circuit.

TWO NEW PARALLEL PROCESSORS FOR REAL TIME CLASSIFICATION OF

3-D MOVING OBJECTS AND QUAD TREE GENERATION

by

FARJAM MAJD

A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
ENGINEERING: COMPUTER AND ELECTRICAL
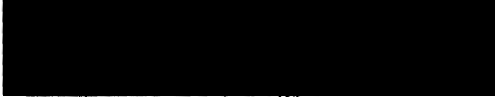
Portland State University

1985

TO THE OFFICE OF GRADUATE STUDIES AND RESEARCH:

The members of the Committee approve the thesis of Farjam Majd presented November 14, 1985.

Dr. F. Badi'i, Chairman

Dr. J. Heneghan

Prof. J. Riley

Dr. M. Ghafarzadeh

APPROVED:

Dr. Pieter A. Frick, Head, Department of Electrical Engineering

Dr. Jim F. Heath, Dean, Graduate Studies and Research

ACKNOWLEDGMENTS

It is my pleasure and duty to express my sincere gratitude to my advisor Dr. Faris Badi'i, who not only guided me through the course of this thesis by giving valuable suggestions and ideas, but also was a source of knowledge and support throughout my master's degree program.

I would like to give special thanks to my parents for their support during the entire course of my education. Without their encouragement my higher education would not have been possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

FIGURE   PAGE

CHAPTER I

INTRODUCTION

Real time image processing is a relatively new field which was made possible primarily because of the advances in IC technology. This has made the implementation of special purpose processors more feasible due to lower cost and more accessible service [1-3]. Image processing of any kind generally involves a large number of operations due to the abundance of information in the data source (image) and its intricate nature. Hence, single processor digital computers do not lend themselves well to real time image processing, and special purpose parallel processors must be designed which reduce the computation time by performing the required operations in parallel instead of in sequence, as in the case of common computers [4, p. 209].

In this thesis, two new parallel processors are presented. In Chapter II, the method of "Invariant Numerical Shape Modeling" (INSM) which was originally developed for 2-D alphanumeric character recognition [5] is extended for 3-D object recognition. The INSM method is not dependent on the size, orientation and location of the object. Subsequently, a bit slice SIMD architecture is proposed for 3-D object recognition and classification.

In Chapter III, the quad tree algorithm, which is normally used for image representation and compaction, is briefly described and a tree architecture is developed based on this algorithm. This processor has a

recursive combinational structure that processes all bits of a picture (pixels) simultaneously and therefore is extremely fast.

CHAPTER II

REAL TIME CLASSIFICATION OF 3-D MOVING OBJECTS

There are two general types of pattern recognition: discriminant
and syntactic [6]. In discriminant methods of pattern recognition, a
set of characteristic measurements, called features, is extracted from
the image. The set of these features is called a feature vector. The
feature vector describes the image under consideration and can be used
for its recognition. Syntactic methods, however, deal with the
components of an image. These components, often called primitives,
compose each pattern which is recognized according to some rules for
parsing the pattern structure.

In this thesis a syntactic recognition is chosen, as described
next.

THE "INSM" ALGORITHM

One of the methods utilized for shape description in syntactic
pattern recognition is the generation of a numeric code or number string
based on the boundaries of an object [5, 7, 8]. INSM is one such
algorithm. This algorithm uses the polygonal representation of the
boundary of the object to generate the code. The coding starts from an
arbitrary node, and the polygon is traversed in a cw or ccw direction,
to one node beyond the starting point. The code is generated based on
the relative direction and length of the line segments (polygon

sides). There are six possible relations between two consecutive line segments based on these criteria. A line segment can either turn to the right or left and be shorter, equal, or longer with respect to the line segment before it. To each one of these six possible configurations, an integer code is assigned. As the polygon is traversed, one such integer is assigned to each line segment. Thus, an n-digit code is used to represent an n-sided polygon. This code is normalized by circularly rotating the number string and selecting the position which yields the largest numerical value. The normalized code is independent of the orientation of the object and the starting node. This is true because the image polygons are closed and the order of the sides traversed are independent of the starting point.

The selected integer codes for INSM are summarized in Table I. The following example illustrates this algorithm.

TABLE I

INTEGER CODES FOR INSM ALGORITHM

Relative Condition Between Line Segments	Integer Code
RIGHT	
Shorter	1
Equal	2
Longer	3
LEFT	
Shorter	4
Equal	5
Longer	6

EXAMPLE 1

Let us code the airplane image shown in Figure 1.

Figure 1. The polygonal representation of an airplane profile.

The coding is done starting from the circled node:

1. right, equal: 2   4. right, longer: 3

2. left, longer: 6   5. right, shorter: 1

3. right, shorter: 1   6. right, shorter: 1

So the object code is 261311, which is normalized to 613112.

Extension to 3-D

In general, a 3-D object can be specified by a number of 2-D projections obtained at different spatial angles around the object. The more complicated the shape of the object is, the more 2-D projections (or profiles) are needed to specify it. Based on this property, a 3-D object can be identified, not by a single object code, but rather by a set of object codes, each one of which is the object code of one of the profiles. One possible feature of profiles is what is called a spike. A spike results from a 180 direction change of a line segment with respect to its predecessor. Thus, in the case of a spike, there are three possible conditions: shorter, equal and longer.

Normally, spikes do not occur as often as other conditions. For this reason, two-digit integer codes were selected for assignment to various spikes. Integer 7 signifies the existence of a spike; and 71, 72, and 73 designate shorter, equal, and longer conditions. These conditions are shown in Figure 2.

Figure 2. Spikes, illustrating (a) shorter, (b) equal, and (c) longer conditions.

The INSM integer codes, including the spike conditions, are

summarized in Table II. The integer codes 1 through 7 are represented internally in a computer using binary codes 001 through 111, respectively. Thus, for every integer code we need three bits, and for a spike we need six. However, spikes occur much less frequently than other conditions. Therefore, for an n-digit object code, we need a memory space on the order of 3n bits. Had we chosen distinct integers for the spikes, such as 7, 8, 9 instead of 71, 72, 73, we would have needed a memory space on the order of 4n bits since 8 and 9 require at least four binary bits. Thus, this selection of spike codes reflects approximately a 25% savings in memory space.

TABLE II

INTEGER CODES FOR INSM ALGORITHM WITH SPIKES

Relative Condition	Integer Code
RIGHT	
Shorter	1
Equal	2
Longer	3
LEFT	
Shorter	4
Equal	5
Longer	6
SPIKE	(7)
Shorter	71
Equal	72
Longer	73

Properties of 3-D INSM

The 3-D INSM algorithm is dependent on the direction of traversal. However, there is a one-to-one relation between cw and ccw traversals, which is depicted in Figure 3. This figure demonstrates that the condition "right, longer" in cw traversal is equivalent to "left, shorter" in ccw traversal (Figure 3a) and vice versa (Figure 3b). The conditions "right, equal" and "left, equal" are equivalent in cw and ccw traversals, respectively. These relations are presented in Table III in terms of integer codes.

(a)   (b)   (c)

Figure 3. Relation between cw and ccw traversals. In cw traversal (a), the condition "right, longer" is equivalent to "left, shorter"; (b) "right, shorter" is equivalent to "left, longer"; and (c) "right, equal" is equivalent to "left, equal" when traversing in ccw direction.

TABLE III

RELATION BETWEEN INSM CODES FOR CW AND CCW TRAVERSALS

CW TRAVERSAL	CCW TRAVERSAL
1	6
2	5
3	4
4	3
5	2
6	1
71	73
72	72
73	71

Table III is used in order to convert a cw code to ccw or vice versa. However, this is not sufficient since not only the codes change when the direction of traversal changes, but also the order of traversal is reversed. Hence, Table III is used to convert the integer codes and then the converted code is written in reverse order to give the correct object code. This property is illustrated in the following example.

EXAMPLE 2.

Suppose we are given the image in Figure 4:

Figure 4. Image for coding example 2.

The cw traversal results in 16131, which is normalized to 61311; and the ccw traversal results in 46166, which is normalized to 66461. Using Table III, 11316 is obtained for the ccw code. Now, writing this in reverse order results in 61311, which is the same as the normalized cw code.

Another one of the properties of the 3-D INSM is the relation between the cw traversal of an image polygon and the ccw traversal of its flip-image polygon. The flip-image is obtained by a 180° rotation of a 2-D image about an arbitrary axis in its plane. This relation is simply that left direction in an image polygon is equivalent to the right direction in the flip-image polygon. This relation is given in Table IV.

TABLE IV

RELATION BETWEEN CW TRAVERSAL OF AN IMAGE AND
CCW TRAVERSAL OF ITS FLIP-IMAGE

CW Codes of Image	CCW Codes of Flip-Image
1	4
2	5
3	6
4	1
5	2
6	3
71	71
72	72
73	73

From the relations given in Tables III and IV, it is clear that the

code and flip-code (code of its flip-image) of an image contain the same information and one can be obtained from the other. This is one of the major strengths of the 3-D INSM algorithm, since it allows the elimination of 50% of profiles otherwise needed for the unique identification of a 3-D object. However, since in coding an image, the traversal is always done in cw or ccw direction, a relation should be derived between cw or ccw traversal of an image and its flip image. Such relation may easily be derived by combining Tables III and IV. The result is presented in Table V.

TABLE V

RELATION BETWEEN CW (CCW) TRAVERSAL OF AN IMAGE
AND ITS FLIP-IMAGE

CW Codes of Image	CW Codes of Flip-Image
1	3
2	2
3	1
4	6
5	5
6	4
71	73
72	72
73	71

A PATTERN RECOGNITION SYSTEM BASED ON THE 3-D INSM ALGORITHM

The aforementioned properties of the 3-D INSM algorithm and its relative simplicity make it a desirable algorithm for pattern recognition and classification. The recognition scheme presented in

this thesis consists of a library of profiles of one or more given objects (a set of six airplanes are used here as an example) coded using this algorithm and the INSM code of an external object which is to be recognized. The code for the external object is generated by a processor from an image of the object, obtained via some data acquisition system. This code is then compared to all library codes to establish its identity.

Hardware Configuration

The hardware arrangement for the realization of this system is shown in Figure 5. The image of the object targeted for identification is acquired by an image acquisition system and fed into a dedicated microprocessor for coding. The code is then sent to an image classifier for identification. The classifier has an SIMD architecture, designed in bit slice fashion. A more detailed diagram of this system is shown in Figure 6. The image classifier consists of a set of library modules, designated Li, where i is a numeric subscript. Every Li chip has three terminals, A, B, and C. Terminal A consists of eight bits for data input; terminal B is used as a clock pulse for data multiplexing; and terminal C is the output signal signifying the identification of the image (input data). The internal structure of the Li chips is shown in Figure 7. The numbers shown in this figure are purely for illustration purposes and may vary according to application and cost considerations. Here, for the sake of illustration, it is assumed that coding is done using an 8-bit microprocessor. It is further assumed that a resolution of 32 digits is sufficient for codes to be unique. Thus, each profile

Figure 5. Block diagram for the pattern recognition system.

14

Figure 6. Connections of the microprocessor with library modules.

Figure 7. Detailed structure of a library chip.

chip consists of 1-13 counters whose clock pulse is provided through terminal B, a 1 x 16 decoder, twelve 8-bit registers for holding the target code, and eighteen 96-bit registers for storing the library profiles. Signal A1 shown in Figure 6 signifies the identification of the one airplane whose profiles are in chips L1 and L2. The entire system for the identification of six airplanes is shown in Figure 8. Each block with output Ai has a structure identical to that shown in Figure 6, but with profiles of different airplanes. Every such block is connected to the μP via the A and B terminals of its Li chips. Signals N, U, and M correspond to nonidentification, unique identification, and multiple identification, respectively.

Functional Description of the Hardware

Let t_i, l_{ij}^k, and C_k be the ith bit of the code register, the ith bit of the jth library register of the L_k module, and the C terminal of the L_k module, respectively. And furthermore, let A_r be the output of the rth block as shown in Figure 8. Where, for the case under consideration, i = 1, 2, . . . , 96; j = 1, 2, . . . , 18; k = 1, 2; and r = 1, 2, . . . 6, then we can write:

Figure 8. Complete pattern recognition system for airplane identification.

$$C_k = \sum_{j=1}^{n_1} \left[\prod_{i=1}^{n_2} \left(t_i \odot 1_{ij}^{k} \right) \right] \ ; \ K = 1, 2 \tag{1}$$

$$A_r = \sum_{k=1}^{n_3} C_k \ , \ r = 1, 2, \ldots, 6 \tag{2}$$

$$M = \sum_{k=1}^{n_3 - 1} \left[A_k \left(\sum_{r=k+1}^{n_3} A_r \right) \right] \tag{3}$$

$$U = \overline{M} \sum_{r=1}^{n_3} A_r \tag{4}$$

$$N = \overline{\left[\sum_{r=1}^{n_3} A_r \right]} \tag{5}$$

where:

n_1 = # of library registers per Li module (18, in our case)

n_2 = # of bits per library register (96)

n_3 = # of distinct objects in the library (6)

Thus, in order to expand the library, we can take advantage of the expandable bit slice design and add Li chips for increasing the number of profiles per object without having to alter the internal structure of the Li chip itself. And for adding more objects, we can add entire modules as the one in Figure 6. To add these Li chips or modules to the system, it is only necessary to use trivial external gating of these subsystems; and no major modification is necessary. Equations (1) - (5) facilitate this expansion.

System Operation

Once a profile is acquired from the target object, it is sent to the microprocessor for coding. Then the code is transmitted to the Li modules eight bits at a time, as shown in Figure 6. In Figure 7, the counter-decoder combination form a multiplexing circuit in conjunction with the tri-state buffers at the inputs of the code registers. The microprocessor sends the first eight bits of the 96-bit code and then clocks the counter via terminal B to enable the input to the first code register by enabling the first output of the decoder. Then the next eight bits are put on the bus and another pulse enables the second decoder output which in turn enables the current data into the second code register, etc. When all 96 bits of the code are loaded into the

code registers, the thirteenth decoder output enables **all** of the code registers' outputs at once, which causes the target profile to be compared to all eighteen library profiles simultaneously. If one or more of the library profiles matches the target profile, then the output at terminal C goes high, signifying this fact. In addition, if the C output of one or more Li modules is high, then signal Ai will go high, as shown in Figure 6. This means airplane Ai has been identified. If exactly one Ai signal is high, then output U will go high, indicating a unique identification. If more than one Ai signal is activated, then signal M will go high, which signifies a multiple identification. This signal means more profiles are needed for unique identification. Signal N will be activated if none of the Ai signals is high, which means that the target was not recognized.

RESULTS

A computer program was written in C language to test the INSM algorithm. The actual profiles of a set of six airplanes [7] were used to carry out this test. The program was successful in establishing the reliability of this method by properly identifying arbitrary objects and random profiles of these airplanes. A Haming distance of zero was chosen for comparison. A copy of this program in provided in Appendix A. As mentioned earlier, due to its flip and orientation invariance, the 3-D INSM algorithm can reduce the number of profiles needed for identification by more than 50%. Actually, in the example used, more than 60% of images provided were discarded because of redundancy with

respect to this algorithm. The profiles used in the test are given in Appendix B.

CONCLUSION

In this paper it was shown that the INSM algorithm is a suitable method for 2-D pattern recognition. This algorithm was extended to 3-D; and after establishing its properties, it was shown, using a computer program, that it can be easily applied to the problem of 3-D pattern recognition and classification, as well. A real time, microprocessor-based pattern recognition system was proposed and developed which consists of an image acquisition system and a microprocessor for generating the INSM code. A parallel processor was designed with a bit slice SIMD architecture for the recognition and classification of a given target image. A set of Boolean equations was derived to facilitate hardware description and expansion. In all, in this thesis a new real time pattern recognition algorithm, complete with an appropriate hardware system, was developed.

CHAPTER III

REAL TIME GENERATION OF QUAD TREES

Generally, digital images are stored in computer memory using a 2-D array. However, this is not an efficient method of data storage since the memory space required increases as n^2 for an n x n image. For this reason, considerable work has been done in the area of digital image representation and compacation. Traditionally, two distinct approaches have been used for this purpose. In the first approach, boundaries of an image have been used for its representation [9, 10]; and in the second, distinct internal regions of the image have been utilized. More recently, other approaches to image representation have been used. Among these are hierarchical data structures such as image pyramid [11, 20] and quad tree [13-16].

In this thesis, the quad tree data structure is used for developing a real time image processor for image representation and compaction.

QUAD TREES

The quad tree is a recursive data structure which is suitable for the representation of 2-D data such as images. One of the advantages of using quad trees is their ability to represent images in a more compact form [17].

A picture frame which has a 2^n x 2^n dimension, will have an (n + 1)-level quad tree representation [8]. A quad tree is generated by

dividing the entire picture frame into four equal quadrants, if all pixels in the whole frame do not have the same gray level (a frame of a single solid color). Then each quadrant is examined separately. If pixels in a given quadrant all have the same gray level, then that quadrant is left untouched and forms a leaf of the tree; otherwise, it forms an internal node. If, however, some pixels have a different gray level than others, then that quadrant is subdivided into four subquadrants. This process is repeated until all subquadrants have only one gray level within themselves. Then each such subquadrant forms one leaf of the quad tree. All the information necessary to reconstruct the image will be contained in the leaves of its quad tree. This quad tree has n + 1 levels, with level 0 corresponding to the root of the tree or the whole picture frame and level n corresponding to a single pixel. Each subdivision corresponds to going one level down in the tree. Figure 9 shows an example of a quad tree. At each level, the correspondence between quadrants and the tree nodes is given by Figure 9c. Other assignments are possible as well. One way of representing a leaf of a quad tree in compact form, at any level, is to store the address of the upper left pixel of that quadrant, the length of its side in terms of pixels, and the gray level of the quadrant in a table. Thus, a quad tree can be stored and represented in this tabular form without storing information about a node's parents and children. Such information may be extracted from this table if needed. An example of such a table is given in Figure 9d.

y 1 2 3 4

x

1 2 3 4

(a)

1 2
3 4

(c)

ROOT

1 2 3 4

1 2 3 4   1 2 3 4

(b)

START ADDRESS	LENGTH OF SIDE	GRAY LEVEL
x=1, y=1	1	BLACK=1
1, 2	1	WHITE=0
2, 1	1	O
2, 2	1	1
1, 3	1	O
1, 4	1	1
2, 3	1	O
2, 4	1	O
3, 1	2	O
3, 3	2	1

(d)

Figure 9. Examples of (a) digital image, (b) its quad tree, (c) correspondence between image quadrants, at any level, and quad tree branches, and (d) tabular representation of the quad tree.

A REAL TIME IMAGE COMPACTION SYSTEM USING QUAD TREES

As was pointed out earlier, quad trees are suitable data structures for compact storage of images. The system proposed here is for the real time compaction of images which are acquired frame by frame through an image acquisition system. This is done using a tree architecture based on the quad tree structure. The quad tree of an image is generated in real time as fast as data can be input to the processor. A block diagram of this system is shown in Figure 10.

Hardware Configuration

The system shown in Figure 10 consists of an image acquisition system, a tree processor, a secondary processor, and a main memory unit. A picture frame is fed into the tree processor, pixel by pixel, via the image acquisition system. Actually the operation of the tree processor is completely independent of the way that data are input to it. The data may be given to the tree processor pixel by pixel, row by row, column by column, all at once, etc. The tree processor itself consists of a combinational circuit called a tree generator, two arrays of RAM, one double arrray of ROM, and another single array of ROM.

Tree Generator. A detailed diagram of the tree generator for a $2^2 \times 2^2$ data array is shown in Figure 11. It consists of a recursive structure beginning with groups of four pixels and expanding outward with groups of four groups, etc. This recurisve structure is illustrated in Figure 12. The letter "R" designates a single register or pixel. These groups are formed by grouping together four units in a

Figure 10. Image compaction system.

Figure 11. Detailed structure of the tree generator.

Figure 12. ,The recursive structure of the tree generator.

square type of structure, forming a larger unit. Then this larger unit is grouped with three other similar units again in a square type of structure to form an even larger unit. This upward recursive procedure is repeated until the desirable height is reached. The largest such unit corresponds to the entire picture frame, or equivalently, to the root of the quad tree; and the smallest unit is a single register which corresponds to the lowest level of the tree, or a single pixel. Thus, the hardware structure of the tree generator is identical to that of a quad tree.

At any level, the units in a given quadrant are connected via an X-NOR-gate, and the outputs of these are fed into an AND-gate. Each such AND-gate also receives the outputs of the AND-gates of its subquadrants, as inputs, and its own output is fed into the inputs of its superquadrant's AND-gate. The output of every AND-gate is also connected to inputs of a non-inverting tri-state buffer and an inverting tri-state buffer. The outputs of these buffers are used for the control of data paths between the tree generator and memory arrays of the tree processor. The enable signal for these buffers, at any given level and a given quadrant, comes from the output of the inverting buffer of their superquadrant. It is important to note that every X-NOR-gate in Figure 11 actually represents eight X-NOR-gates, one per register bit. Thus, their input lines represent eight lines, while all other lines are just single lines.

Memory Structure of Tree Processor. In addition to the tree generator, the tree processor has three distinct arrays of memory

elements. These include an array of RAM called the G-RAM, another RAM array called the L-RAM, a double array of ROM designated X-ROM and Y-ROM, and an array of ROM called L-ROM.

The G-RAM, L-RAM, X-ROM, and Y-ROM have 2^n x 2^n registers each, one for every register in the tree generator. The L-ROM has n + 1 registers. There is a row-by-row correspondence between the registers in the tree generator and the registers in these four memory arrays. The registers in each array are numbered according to row and column numbers of the pixels in the tree generator, as illustrated in Figure 11 for the 2^2 x 2^2 case.

The G-RAM, L-RAM, X-ROM, and Y-ROM are for storing the gray level, the length of side of a quadrant of the quad tree, the row number, and the column number of their corresponding registers in the tree generator, respectively. The L-ROM contains the n + 1 possible lengths of sides for quadrants at different levels of the tree.

Processor. When processing multiple frames of pictures, we need a larger memory than that available in the tree processor since that memory is for storing one frame of picture (in quad tree representation) only. Thus, after a frame of picture is processed, it is read from the memory of the tree processor into a larger memory space. This is done by the processor. Data compaction actually takes place during this transfer.

Main Memory. The main memory is a mass storage device for storing quad trees in compact tabular form after they are generated in the tree processor. As mentioned above, the quad trees are transferred

to the main memory using the processor.

System Operation

The image to be processed is loaded into the tree generator as shown in Figure 10. The tree generator works in two combinational passes. In the first pass, as the registers are loaded, starting with the lowest level (the register level), if pixels in a group have the same gray level (same bit pattern), then the AND-gate of that group will have a high output. This output signifies that this quadrant may be a leaf of the tree, if it is not part of a bigger quadrant at a higher level in the tree. If all the AND-gates in the next upper quadrant containing this group are high, and if, in addition, the pixels in all similar groups in this quadrant have the same gray level, then the AND-gate of this quadrant will go high, indicating that all pixels in this quadrant are the same. This upward recursive process takes place simultaneously in all quadrants of the tree, starting with the lowest level and combinationally propagating up to the root of the tree. The second pass is a downward recursive process from the root of the tree down to the lowest level. During the second pass at any level in the tree, starting with the top level, the output of the AND-gate of a given quadrant is either high or low. If it is high, it signifies that all pixels in that quadrant are the same and hence, that quadrant is a leaf in the tree. However, if it is low, it indicates that all pixels in that quadrant are not the same, and it has to be divided into four subquadrants. This division is done using the signal now generated at the output of the inverting buffer of this AND-gate. This signal

propagates down to enable the buffers of the AND-gates of all its subquadrants. Depending on the states of their respective AND-gates, these subquadrants either form some leaves of the quad tree, or propagate the signals from the outputs of their inverting buffers to the next level down. This process continues in all quadrants simultaneously and through all levels (with a combinational propagation delay) until a timer set for the maximum propagation delay of both passes terminates the second pass by reseting all registers in the tree generator to zero.

Before the termination of the second pass, the quad tree structure is specified by the outputs of the non-inverting and inverting tri-state buffers in every level. But these are actually only the control signals which will subsequently generate the quad tree table in the memory arrays. There are one or more sets of eight tri-state buffers at the input of each register in the G-RAM and L-RAM arrays, as shown in Figure 13. The input to each register in the G-RAM comes from the output of the corresponding register in the tree generator, and the inputs for the L-RAM come from the L-ROM. The tri-state buffers' enable lines are connected to the control signals from the tree generator. These control signals are shown for a $2^2 \times 2^2$ system as circled numbers in Figures 11 and 13. The control signals consist of the outputs of either non-inverting or inverting buffers. The high outputs of non-inverting buffers constitute control signals at any level of the tree because a high signal at their output indicates that their respective quadrants are leaves of the tree and are ready for tabulation. However, a high signal at the output of the inverting buffers constitutes control

Figure 13. Interconnections of G-RAM, L-RAM, L-ROM and control signals.

signals only at the lowest level, where the last buffers appear, and the lower level is the register level. Such signals at higher levels would enable the tri-state buffers of their respective subquadrants, as explained earlier. If a control signal comes from a non-inverting buffer, it will only enable the input of those registers in the G-RAM and L-RAM arrays that corresponds to the upper left register in its quadrant. This is because all pixels in that quadrant have the same gray level, and the upper left register serves as a representative of the gray level of the entire quadrant. The address of this register is called the start address of this quadrant. However, if a control signal comes from an inverting buffer, then it means that it is at the lowest level and that at least two of the pixels in this lowest level quadrant have different gray levels and should be tabulated separately. Thus, such signal will enable the inputs to all registers of the G-RAM and L-RAM arrays which correspond to the four pixels in its quadrant. Based on these properties, the connections from the control signals to the input enables of the G-RAM and L-RAM can be easily established. It is worthwhile to observe that control signals have either four connections (for inverting buffers) or one connection (for non-inverting buffers). The control signals have identical connections to G-RAM and L-RAM arrays.

In order to determine the precise connections from the L-ROM to L-RAM, we need to find a general relationship between the length of side of a quadrant in a given level and all possible registers which can possibly have that length of side in a quad tree structure. Since a

quad tree is formed by successively dividing each side of its quadrants

by 2, the starting address of each has to be an integer power of 2.

Also, at every level there will be i x i such starting points, where i-1

is the level number, starting with zero for the top level or root of the

tree. Using these properties, the following equations and algorithm A

were derived. For a 2^n x 2^n array we have:

$$x = 1 + j * 2^k; \quad j = 0, 1, \ldots, \frac{2^n}{2^k} - 1 \tag{6}$$

$$y = 1 + 1 * 2^k; \quad 1 = 0, 1, \ldots, \frac{2^n}{2^k} - 1 \tag{7}$$

where:

$$k = \log_2 [\text{length of side}] = 0, 1, \ldots, n \tag{8}$$

$$x = \text{row number of the start point}$$

$$y = \text{column number of the start point.}$$

Algorithm A

```
        For k = Ø to n [
                            n
            Set:  P = 1, Q = 2   -1
                           ___
                            k
                           2

                For j = Ø to Q [

                    For 1 = Ø to Q [
                                        k              k
                        Set:  X(P) = 1 + j * 2 , y(P) = 1 + 1 * 2
                        Set:  P = P + 1
                    ]
                ]
        ]
```

Equations (6) and (7) were derived to be used in algorithm A. This algorithm gives the correspondence between any given possible length of side, 2^k, for k = 0, 1, . . ., n, in a 2^n x 2^n array, and the start address (upper left pixel), (x, y), of all subquadrants which could possibly have that length of side. Thus, for k = 0, algorithm A will produce all pixels in the array since $2^k = 2^0 = 1$ and all pixels can be the start address of a subquadrant with a length of side of 1 (a single pixel). And for k = n, the algorithm will only produce (x, y) = (1, 1), because only pixel (1, 1) can be the start address of a subquadrant with a length of side of 2^n (the entire array).

The L-ROM contains all possible lengths of side, 2^k, for k = 0, 1, . . . , n. Hence, for each value of k, the coordinates of the L-RAM registers which can have a side of length 2^k, are generated using algorithm A, and connections are made accordingly. The address of all registers of the tree generator are stored in the X-ROM and Y-ROM arrays since they are known in advance. Once the value of a register and the length of side of the quadrant it belongs to are transferred to the appropriate arrays, the table entry for that leaf is complete. It should be noted that not all registers in the G-RAM and L-RAM arrays will contain a value for a given quad tree since a quadrant which is a leaf will be represented only by its start register, and the rest of its registers will be ignored. Thus, although the memory arrays of the tree processor will contain the quad tree table, it is not any more compact than the original picture stored in the tree generator because of the gaps in these memory arrays. Hence, to compress this table and at the

same time transfer it to the main memory, a processor is used, as shown in Figure 10. This processor first checks the L-RAM, one register at a time. If it was non-zero, then it transfers the entire table entry corresponding to that register (i.e., the contents of L-RAM and the corresponding registers in G-RAM, X-ROM, and Y-ROM) to the main memory. If it was zero, however, the processor ignores that entire table entry since it is actually a blank line. This way, the filled table entries are stored one after another, without any gaps, in the main memory. After the last entry is processed, the processor will send a signal to reset all registers in the G-RAM and L-RAM arrays to zero. Now the entire system is ready to process another frame of picture. While the processor is busy reading the table entries, the tree generator can receive the next picture frame since it has already been reset after the second pass of the previous image. Therefore, pipelining has been incorporated into this system as well, for maximum efficiency.

CONCLUSION

A new real time image compaction system was developed based on the quad tree data structure. It consists of a tree architecture which processes 2-D data in parallel and is independent of the data inputing process. It was proposed that this system is best suited for the processing of multiple frames of pictorial data in real time since it combines parallel processing with pipelining for maximum speed. Formulas were derived to facilitate the specification of hardware connections.

REFERENCES

1. D. Agrawal and R. Jain, "A Pipelined Pseudoparallel System Architecture for Real Time Dynamic Scene Analysis," _IEEE Trans on Comps._, Vol c-31, No. 10 (October 1982).

2. F. A. Briggs, K. S. Fu, K. Hwang, and J. Patel, "PM: A Reconfigurable Multiprocessor System for Pattern Recognition and Image Processing," _Proc. NCC, AFIPS_ (June 1979).

3. L. S. Davis, "Computer Architecture for Image Processing," _Proc. Picture Data Descrip. Manag Conf._ (August 27-28, 1980).

4. John P. Hayes, _Computer Architecture and Organization_, McGraw-Hill, New York (1978).

5. F. Badi'i and B. Peikari, "Invariant Numerical Shape Modeling," _Proc. IEEE Conf. on Computer Vision and Pattern Recognition_, Washington, D. C., (June 1983).

6. K. S. Fu, "Recent Advances in Syntactic Pattern Recognition," School of Electrical Engineering, Purdue University, W. Lafayette, Indiana.

7. T. Wallace and O. Mitchell, "Real Time Analysis of Three Dimensional Movements Using Fourier Descriptors," School of Electrical Engineering, Purdue University, W. Lafayette, IN.

8. Thes Pavlidis, _Algorithms for Graphics and Image Processing_, Computer Science Press, Rockville, MD. (1982).

9. H. Freeman, "On the Encoding of Arbitrary Geometric Configurations," _IRE Trans. on Electronic Computers_, Vol. EC-10, No. 2, pp.260-268 (1970).

10. V. Montanari, "A Note on Minimal Length Polygonal Approximation to a Digitized Contour," _Comm. of ACM_, Vol. 13, No. 7, pp. 41-47 (1970).

11. M. D. Levine and J. Leamet, "A Method for Non-Purposive Picture Segmentation," _Proc. of 3rd Int. J. Conf. on Pattern Recognition_, pp. 474-497 (1976).

12. S. Tanimoto and A. Klinger, "Structured Computer Vision," Chs. 2-3, _Academic Press_ (1970).

13. A. Klinger and C. R. Dyer, "Experiments in Picture Representation Using Regular Decomposition," Computer Graphics and Image Processing, Vol. 5, pp 68-105 (1976).

14. H. Samet, "Region Represntation: Raster-to-Quad Tree Conversion," IEEE Trans. on Pattern Analysis and Machine Intelligence, Vol. 3, pp. 93-95 (1981).

15. H. Samet, "Region Representation: Quad Tree From Binary Array," Computer Graphics and Image Processing, Vol. 13, pp. 88-93 (1980).

16. G. M. Hunter and K. Steiglitz, "Operation on Images Using Quad Tree," IEEE Trans. on Pattern Analysis and Machine Intelligence, Vol.1, No. 2, pp. 145-153 (1979).

17. C. H. Chien and J. K. Aggarwal, "A Normalized Quad Tree Representation," Proc. IEEE Conf. on Computer Vision and Pattern Recognition, p. 121 (June 1983).

APPENDIX A

A COMPUTER PROGRAM FOR TESTING THE 3-D INSM ALGORITHM

```
/***********************************************************************/
/* This is a function which *           ARGUMENTS :                  */
/* implements a pattern      *          ***********                  */
/* recognition algoritnm     * x_image = x_coordinate of points      */
/* created by Dr. Faris      * y_image = y_coordinate "    "         */
/* Badi'i. It was written by* code = array containing the code       */
/* Farjam Majd on 5-11-1984.* flip_code = array for the flip_code    */
/*                           * n = # of code digits                  */
/*                           *                                       */
/***********************************************************************/

#include<math.h>

code_gen(x_image,y_image,code,flip_code,n)
double x_image[],y_image[];
int code[],flip_code[],n;
{
    double px,py,qx,qy,r,dl;    /* define edge vectors */
    int i,j;
    for(i = 0 , j = 0 ; i < n ; ++i , ++j)
    {
        px = x_image[j+1] - x_image[j];
        py = y_image[j+1] - y_image[j];
        qx = x_image[j+2] - x_image[j+1];
        qy = y_image[j+2] - y_image[j+1];
        r = px*qy - py*qx;          /* calculate rz of resultant vector, r */
                                    /* calculate length difference */
        dl = sqrt(pow(px,2.0) + pow(py,2.0)) - sqrt(pow(qx,2.0) + pow(qy,2.0));
        if(r < - 0.00001)               /* right turn on condition true */
        {
            if(dl < - 0.00001)          /* longer on condition true */
            {
                code[i] = 3;
            }
            else if(dl > 0.00001)       /* shorter on condition true */
            {
                code[i] = 1;
            }
            else                        /* equal */
            {
                code[i] = 2;
            }
        }
        else if(r > 0.00001)            /* left turn */
        {
            if(dl < - 0.00001)          /* longer on condition true */
            {
                code[i] = 6;
            }
            else if(dl > 0.00001)       /* shorter on condition true */
            {
                code[i] = 4;
            }
            else
            {
                code[i] = 5;            /* equal */
            }
        }
        else
        {
            code[i] = 7;                /* spike */
            if(dl < - 0.00001)
            {
```

```
                }
                else if(d1 > 0.00001)
                {
                        code[i+1] = 1;
                }
                else
                {
                        code[i+1] = 2;
                }
                ++i;
        }
    }
    for(i = 0 ; i < n ; ++i)                 /* generate flip_code */
    {
            if(code[i] == 1)
            {
                    flip_code[n-i-1] = 3;
            }
            else if(code[i] == 2)
            {
                    flip_code[n-i-1] = 2;
            }
            else if(code[i] == 3)
            {
                    flip_code[n-i-1] = 1;
            }
            else if(code[i] == 4)
            {
                    flip_code[n-i-1] = 6;
            }
            else if(code[i] == 5)
            {
                    flip_code[n-i-1] = 5;
            }
            else if(code[i] == 6)
            {
                    flip_code[n-i-1] = 4;
            }
            else if(code[i] == 7)
            {
                    flip_code[n-i-2] = 7;
                    if(code[i+1] == 1)
                    {
                        flip_code[n-i-1] = 3;
                    }
                    else if(code[i+1] == 2)
                    {
                        flip_code[n-i-1] = 2;
                    }
                    else if(code[i+1] == 3)
                    {
                        flip_code[n-i-1] =1;
                    }
                    ++i;
            }
        }
}

/*************************************************/
/* This is a routine for standardizing the      */
/* code generated by the code_gen routine,      */
/* that is, to shift the code to form the        */
/* largest number.                               */
/*************************************************/
```

```c
int code[],tmp_code[],n;
{
    int temp,i,j,k,l;
    for(i = 0 ; i < n ; ++i)              /* duplicate code */
    {
        tmp_code[i] = code[i];
    }
    for(i = 0 ; i < n ; ++i)      /* shift counter */
    {
        temp = tmp_code[0];              /* shift code to the right */
        for(j = 1 ; j < n ; ++j)
        {
            tmp_code[j-1] = tmp_code[j];
        }
        tmp_code[n-1] = temp;
        for(k = 0 ; k < n ; ++k)    /* digit counter */
        {
            if(tmp_code[k] > code[k])
            {
                for(l = 0 ; l < n ; ++l)
                {
                    code[l] = tmp_code[l];
                }
                break;              /* terminate loop if condition met */
            }
            else if(tmp_code[k] < code[k])
            {
                break;
            }
        }
    }
}


/*****************************************/
/* This is a routine for comparing the codes */
/* generated by the code_gen routine with    */
/* codes of the same kind.                    */
/*****************************************/

compare_code(code,lib,flag,n)      /* n = # of code digits */
int code[],lib[][30],flag,n;       /* lib = library of images */
{                         /* flag = used when there is other than unique lib */
    int i,j,k;
    for(i = 0; i < 174 ; ++i)
    {
    k = 0;
        for(j = 0 ; j < n ; ++j)
        {
            if(code[j] != lib[i][j] || (j == n-1 && lib[i][j+1] != 0))
            {
                break;     /* terminate loop if condition met */
            }
            else if(j == n-1)
            {
                if(i < 29)
                {
                    printf("\n This is a 'Mirage' airplane \n");
                    ++k;
                }
                else if(i >= 29 && i < 56)
                {
                    printf("\n This is a 'Mig' airplane \n");
                    ++k;
                }
                else if(i >= 56 && i < 86)
```

```
                              printf("\n This is a 'Phantom' airplane \n");
                              ++k;
                      }
                      else if(i >= 86 && i < 116)
                      {
                              printf("\n This is a 'F104' airplane \n");
                              ++k;
                      }
                      else if(i >= 116 && i < 145)
                      {
                              printf("\n This is a 'F10t' airplane \n");
                              ++k;
                      }
                      else if(i >= 145 && i < 174)
                      {
                              printf("\n This is a 'B57' airplane \n");
                              ++k;
                      }
                 }
              }
      }
      if(k == 0)
      {
              *flag = 0;
      }
      else if(k > 1)
      {
              printf("\n More images are needed for unique identification \n");
              *flag = 1;
      }
}
/******************************************************/
/* This is a program for testing the 'pat-recl.c' file */
/******************************************************/

#include<math.h>

int lib[174][30] =              /* define library of images */
{
      /*********************/
      /*       MIRAGE       */
      /*********************/

      {6,2,4, 1,3,4, 6,1,3, 4,2,6, 1,3,4, 6,1,3, 0,0,0, 0,0,0, 0,0,0, 0,0,0
      {7,2,4, 6,1,3, 4,6,1, 3,4,2, 6,1,3, 4,6,1, 3,4,0, 0,0,0, 0,0,0, 0,0,0
      {7,2,4, 6,3,6, 1,3,4, 2,6,1, 3,4,1, 4,6,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0
      {7,3,4, 6,7,2, 4,6,7, 2,4,6, 1,3,4, 2,6,1, 3,4,0, 0,0,0, 0,0,0, 0,0,0
      {7,2,4, 6,2,4, 2,2,6, 2,4,6, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0
      {7,2,4, 6,2,4, 2,6,2, 4,6,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0
      {6,2,4, 1,3,4, 6,1,3, 4,3,1, 6,1,3, 4,6,1, 3,0,0, 0,0,0, 0,0,0, 0,0,0
      {6,1,4, 6,1,3, 4,6,1, 3,4,1, 6,1,3, 4,6,1, 3,0,0, 0,0,0, 0,0,0, 0,0,0
      {7,2,4, 6,3,6, 1,3,4, 1,6,1, 2,4,6, 2,4,6, 0,0,0, 0,0,0, 0,0,0, 0,0,0
      {7,3,6, 7,1,4, 6,7,1, 6,6,7, 3,4,1, 6,1,3, 4,6,0, 0,0,0, 0,0,0, 0,0,0
      {7,3,4, 1,4,6, 3,4,1, 1,6,6, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0
      {7,3,4, 1,4,6, 3,1,5, 6,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0
      {7,2,6, 1,6,1, 3,4,6, 1,3,6, 2,4,1, 3,4,6, 1,3,4, 0,0,0, 0,0,0, 0,0,0
      {6,3,4, 3,4,6, 1,3,6, 1,4,1, 3,4,6, 1,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0
      {6,3,6, 1,4,6, 3,4,1, 3,6,1, 4,1,3, 4,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0
      {6,4,1, 3,6,1, 4,1,3, 4,3,6, 1,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0
      {6,3,1, 1,4,3, 4,1,4, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0
      {7,3,4, 1,1,6, 6,3,1, 4,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0
      {6,6,1, 3,6,1, 4,1,3, 4,6,1, 3,4,3, 1,6,1, 3,0,0, 0,0,0, 0,0,0, 0,0,0
      {6,3,4, 6,1,3, 6,1,4, 1,3,4, 4,2,6, 1,4,2, 1,0,0, 0,0,0, 0,0,0, 0,0,0
      {6,1,4, 1,3,4, 1,6,1, 3,4,4, 3,4,1, 3,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0
      {7,2,4, 3,6,1, 4,6,3, 6,1,6, 4,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0
```

```
{6,4,3, 3,6,1, 4,1,3, 4,6,1, 4,3,1, 6,1,3, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{6,1,4, 3,3,4, 3,3,1, 4,1,3, 4,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{6,6,1, 4,3,3, 1,4,3, 1,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{6,1,3, 4,3,3, 1,4,3, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{4,3,3, 1,4,1, 3,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{4,1,4, 1,3,1, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.

/**********************/
/*        MIG         */
/**********************/

{7,2,6, 1,3,4, 6,1,3, 6,2,4, 1,3,4, 6,1,3, 4,0,0, 0,0,0, 0,0,0, 0,0,0}.
{7,2,4, 6,1,3, 4,6,3, 4,2,6, 1,4,4, 1,1,4, 6,0,0, 0,0,0, 0,0,0, 0,0,0}.
{7,2,4, 1,3,4, 6,3,4, 2,6,1, 4,6,1, 3,6,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{7,2,4, 1,4,6, 3,4,2, 6,1,4, 6,3,6, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{7,2,4, 4,3,6, 4,1,3, 4,1,3, 6,1,3, 6,4,1, 4,6,0, 0,0,0, 0,0,0, 0,0,0}.
{7,2,4, 6,3,2, 1,4,6, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{7,2,4, 6,2,4, 3,1,6, 2,4,6, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{6,3,6, 1,3,4, 6,3,4, 2,6,1, 4,4,1, 1,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{6,3,6, 3,4,6, 1,3,4, 3,6,1, 4,6,1, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{6,3,4, 1,3,3, 6,1,4, 4,1,2, 6,2,4, 6,1,3, 4,0,0, 0,0,0, 0,0,0, 0,0,0}.
{6,3,6, 3,4,3, 6,3,4, 1,1,3, 6,3,4, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{7,2,6, 1,1,6, 3,4,4, 3,1,1, 6,6,3, 4,6,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{7,2,4, 4,3,1, 3,1,6, 3,1,1, 4,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{7,2,4, 3,4,6, 1,3,1, 6,6,7, 2,4,3, 1,4,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{6,4,1, 3,4,3, 4,1,3, 4,6,1, 1,6,3, 5,1,6, 3,3,0, 0,0,0, 0,0,0, 0,0,0}.
{6,3,4, 6,3,4, 1,3,4, 1,3,4, 6,1,3, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{6,3,4, 1,1,3, 4,1,4, 6,1,3, 4,3,4, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{7,2,6, 1,1,3, 4,3,4, 6,1,3, 1,6,6, 7,2,4, 4,6,0, 0,0,0, 0,0,0, 0,0,0}.
{7,2,6, 1,6,6, 7,2,4, 3,1,1, 3,4,1, 4,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{6,1,1, 3,4,3, 4,3,4, 3,4,1, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{6,3,4, 3,4,1, 1,3,4, 1,4,6, 1,3,4, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{7,3,4, 4,7,2, 6,1,2, 3,4,1, 6,6,1, 3,4,6, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{7,1,5, 3,4,1, 3,4,6, 3,1,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{6,1,3, 4,3,1, 3,1,4, 3,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{7,2,4, 3,4,3, 1,1,6, 1,1,4, 6,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{4,3,3, 1,4,1, 1,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{7,2,4, 3,1,1, 3,4,1, 3,4,6, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.

/**********************/
/*      PHANTOM       */
/**********************/

{6,6,1, 3,4,2, 6,1,3, 4,4,4, 2,6,3, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{7,2,4, 6,3,4, 6,3,4, 1,6,7, 2,4,3, 6,1,4, 6,1,4, 6,0,0, 0,0,0, 0,0,0}.
{7,3,4, 6,6,1, 4,2,6, 1,4,4, 6,7,1, 5,6,7, 2,4,5, 0,0,0, 0,0,0, 0,0,0}.
{7,2,4, 6,3,4, 6,3,1, 3,1,4, 6,1,4, 6,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{7,2,6, 6,7,2, 4,4,7, 2,6,6, 3,2,1, 4,4,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{6,6,1, 3,4,3, 2,1,6, 1,3,4, 4,1,3, 1,1,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{6,3,4, 6,1,3, 4,3,3, 4,1,3, 4,4,1, 4,2,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{7,1,6, 6,3,4, 4,1,6, 2,6,1, 6,6,3, 4,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{7,2,6, 6,3,4, 3,1,6, 3,6,1, 4,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{7,3,4, 3,4,1, 4,6,7, 2,6,6, 7,3,1, 1,4,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{7,2,6, 3,6,1, 4,6,7, 2,4,6, 7,2,4, 1,4,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{7,2,6, 6,7,3, 4,2,4, 7,2,6, 3,4,1, 4,6,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{6,6,1, 3,4,3, 2,1,6, 1,3,6, 1,1,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{6,1,3, 6,2,4, 1,3,4, 6,1,3, 4,3,4, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{6,6,3, 4,3,3, 4,3,6, 4,1,3, 6,2,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{7,2,6, 6,3,1, 3,6,1, 3,4,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{7,2,6, 7,2,6, 1,3,1, 4,1,4, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{7,3,4, 3,3,1, 4,1,3, 4,6,7, 2,6,6, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{7,3,4, 3,3,3, 4,1,3, 4,6,7, 2,6,4, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{6,5,1, 3,4,3, 2,1,4, 1,3,6, 1,3,1, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{6,6,1, 3,4,3, 4,6,1, 3,4,1, 1,6,1, 2,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
{6,4,6, 3,1,1, 6,1,3, 4,1,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0}.
```

```
{7,3,4, 3.3,1, 6,1,3, 4,7,2, 6,4,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0.
{7,2,6, 3,4,1, 3,6,6, 7,2,4, 2,1,4, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0.
{4,3,2, 3,4,1, 3,3,4, 1,3,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0.
{7,3,1, 1,4,1, 2,6,3, 4,4,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0.
{6,3,4, 3,1,3, 4,1,2, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0.
{4,1,3, 3,3,1, 3,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0.
{4,1,3, 3,1,3, 1,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0.

/*********************/
/*        F104       */
/*********************/

{7,3,7, 1,6,4, 4,3,2, 1,6,6, 4,7,3, 7,1,6, 4,3,3, 1,6,4, 0,0,0, 0,0,0.
{7,3,7, 1,6,4, 2,6,4, 7,3,7, 1,6,4, 1,6,2, 2,3,4, 2,6,4, 0,0,0, 0,0,0.
{7,3,3, 4,5,6, 1,7,1, 6,4,1, 5,6,3, 2,1,4, 5,2,6, 1,0,3, 0,0,0, 0,0,0.
{6,2,2, 2,4,5, 2,6,1, 3,4,2, 4,2,5, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0.
{7,2,4, 4,6,3, 1,4,6, 6,7,2, 4,1,3, 6,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0.
{7,3,7, 1,4,6, 6,7,2, 4,2,6, 7,2,4, 4,6,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0.
{7,3,7, 1,6,4, 4,1,3, 2,1,3, 6,6,4, 7,3,7, 1,6,4, 3,2,1, 6,4,0, 0,0,0.
{7,3,7, 1,6,4, 4,3,1, 1,3,6, 6,4,7, 3,7,1, 6,4,3, 3,6,4, 0,0,0, 0,0,0.
{7,3,7, 1,6,4, 4,6,1, 3,1,3, 4,3,6, 4,7,3, 7,1,6, 4,1,3, 4,6,4, 0,0,0.
{7,3,7, 1,4,6, 2,6,1, 3,4,1, 1,6,1, 3,4,4, 6,0,0, 0,0,0, 0,0,0, 0,0,0.
{7,3,7, 1,4,5, 3,6,7, 3,4,1, 1,6,7, 2,4,4, 6,0,0, 0,0,0, 0,0,0, 0,0,0.
{7,3,7, 1,4,6, 2,6,7, 2,4,1, 6,7,2, 4,3,4, 6,0,0, 0,0,0, 0,0,0, 0,0,0.
{7,3,7, 1,4,6, 6,7,2, 6,1,6, 7,2,4, 3,4,6, 0,0,0, 0,0,0, 0,0,0, 0,0,0.
{7,3,7, 1,6,4, 4,6,1, 3,1,1, 2,6,6, 4,7,3, 7,1,6, 4,3,2, 1,6,4, 0,0,0.
{7,3,7, 1,6,4, 4,6,1, 3,1,1, 3,5,3, 6,4,7, 3,7,1, 6,4,3, 1,1,5, 6,4,0.
{7,3,7, 1,6,6, 4,6,1, 3,1,3, 4,1,3, 6,4,7, 3,7,1, 6,4,3, 3,6,4, 0,0,0.
{7,3,7, 1,4,5, 1,3,3, 4,7,3, 2,4,3, 1,6,4, 7,3,3, 6,4,4, 0,0,0, 0,0,0.
{7,3,7, 1,4,4, 3,6,7, 3,1,1, 4,6,7, 3,4,6, 0,0,0, 0,0,0, 0,0,0, 0,0,0.
{7,3,7, 1,4,4, 3,6,7, 3,1,6, 7,3,4, 4,5,6, 6,0,0, 0,0,0, 0,0,0, 0,0,0.
{7,3,7, 1,4,4, 7,3,1, 6,7,2, 4,3,4, 6,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0.
{7,3,7, 1,6,4, 4,3,6, 1,3,3, 3,4,4, 7,3,7, 1,6,4, 3,2,1, 6,4,0, 0,0,0.
{7,3,7, 1,6,4, 4,4,3, 1,1,3, 4,4,7, 3,7,1, 6,4,3, 3,4,4, 0,0,0, 0,0,0.
{7,3,1, 6,4,4, 3,3,1, 3,1,3, 4,4,7, 3,1,6, 3,1,4, 4,0,0, 0,0,0, 0,0,0.
{7,3,7, 1,4,5, 2,1,5, 3,1,6, 1,4,7, 3,4,6, 7,3,1, 6,4,4, 6,0,0, 0,0,0.
{7,3,7, 1,4,6, 1,2,1, 4,7,2, 4,6,7, 3,6,4, 0,0,0, 0,0,0, 0,0,0, 0,0,0.
{7,3,7, 1,6,6, 3,2,1, 4,4,6, 7,3,7, 1,6,4, 6,4,2, 3,3,3, 4,4,0, 0,0,0.
{7,3,7, 1,6,6, 6,1,1, 5,3,4, 4,7,3, 1,6,4, 7,2,6, 4,5,0, 0,0,0, 0,0,0.
{6,3,2, 3,4,4, 3,1,6, 1,3,1, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0.
{7,2,6, 3,4,3, 1,6,3, 1,4,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0.
{7,1,6, 3,1,1, 3,4,3, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0.

/*********************/
/*        F105       */
/*********************/

{6,4,1, 3,4,6, 1,6,2, 4,3,4, 6,1,3, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0.
{7,2,4, 6,3,4, 6,1,3, 6,5,4, 1,3,4, 6,1,4, 6,0,0, 0,0,0, 0,0,0, 0,0,0.
{7,3,4, 2,6,7, 1,4,4, 7,2,6, 6,7,2, 4,4,7, 2,6,6, 0,0,0, 0,0,0, 0,0,0.
{7,2,4, 6,3,4, 2,2,6, 1,4,6, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0.
{7,2,4, 6,3,4, 2,6,1, 4,6,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0.
{7,2,4, 4,4,3, 7,2,4, 3,6,1, 3,4,1, 6,7,2, 1,6,6, 6,0,0, 0,0,0, 0,0,0.
{6,5,4, 1,3,4, 4,1,3, 6,3,1, 6,3,4, 6,1,3, 0,0,0, 0,0,0, 0,0,0, 0,0,0.
{7,3,4, 1,6,7, 2,6,6, 1,3,4, 6,1,6, 7,1,6, 4,7,3, 4,6,1, 6,0,0, 0,0,0.
{7,1,4, 6,3,4, 1,1,1, 6,1,1, 6,4,3, 6,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0.
{7,3,6, 1,1,6, 3,6,7, 1,6,4, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0.
{7,3,6, 1,4,2, 7,1,6, 4,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0.
{7,3,4, 6,3,4, 6,7,2, 4,3,1, 3,6,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0.
{6,4,1, 3,6,1, 4,1,2, 4,6,1, 4,3,3, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0.
{6,3,6, 1,4,1, 3,4,6, 1,4,2, 3,6,3, 4,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0.
{7,3,6, 1,4,1, 3,4,6, 1,4,3, 4,7,3, 4,4,6, 0,0,0, 0,0,0, 0,0,0, 0,0,0.
{7,1,3, 4,1,3, 4,6,1, 4,3,4, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0.
{7,3,1, 3,4,3, 6,1,4, 4,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0.
{6,1,4, 3,3,1, 3,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0.
```

```
    {7,3,4,  6,7,3,  1,6,7,  1,4,3,  6,1,3,  4,6,1,  4,0,0,  0,0,0,  0,0,0,  0,0,
    {7,3,4,  1,6,7,  1,4,3,  4,1,1,  5,6,1,  1,4,3,  4,0,0,  0,0,0,  0,0,0,  0,0,
    {6,6,1,  3,4,1,  3,1,4,  1,1,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,
    {7,3,1,  3,1,6,  1,4,2,  5,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,
    {6,1,3,  4,2,3,  1,4,1,  1,4,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,
    {6,1,3,  4,2,3,  1,3,4,  1,4,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,
    {6,1,3,  4,2,3,  1,3,4,  3,1,5,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,
    {6,1,3,  4,3,3,  1,1,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,
    {4,1,3,  4,1,3,  1,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,


    /********************/
    /*      B57         */
    /********************/

    {6,4,1,  6,1,3,  4,3,6,  1,2,3,  4,1,6,  1,3,4,  3,6,4,  1,3,1,  3,0,0,  0,0,
    {6,6,6,  1,3,4,  2,6,1,  3,4,4,  4,6,3,  1,3,4,  3,6,1,  2,3,4,  1,6,1,  3,
    {6,3,6,  1,3,4,  6,1,4,  2,6,3,  4,6,1,  3,4,3,  4,5,1,  2,3,4,  0,0,0,  0,0,
    {7,2,5,  6,7,2,  4,6,1,  1,5,4,  2,2,6,  5,3,3,  4,6,7,  2,4,5,  0,0,0,  0,0,
    {6,3,1,  4,3,6,  2,4,1,  6,3,1,  4,2,2,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,
    {7,2,6,  1,1,6,  3,4,3,  3,4,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,
    {7,2,6,  1,4,2,  3,6,3,  4,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,
    {6,6,1,  3,4,3,  6,1,3,  4,4,5,  3,1,3,  1,6,1,  3,4,6,  3,1,3,  1,5,0,  0,0,
    {6,2,6,  3,4,6,  1,3,1,  6,1,1,  5,5,3,  1,3,4,  4,6,2,  2,0,0,  0,0,0,  0,0,
    {7,2,4,  6,3,4,  1,3,6,  1,3,4,  5,3,6,  1,6,0,  0,0,0,  0,0,0,  0,0,0,  0,0,
    {7,3,4,  3,4,1,  6,1,4,  4,3,4,  7,2,4,  6,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,
    {6,3,4,  3,1,6,  3,4,1,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,
    {7,3,4,  1,6,3,  1,6,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,
    {7,3,4,  4,3,1,  3,4,3,  6,4,3,  1,3,4,  5,3,5,  3,3,4,  3,6,4,  2,3,4,  0,0,
    {7,3,4,  6,1,3,  4,1,3,  4,1,3,  4,4,3,  2,3,0,  0,0,0,  0,0,0,  0,0,0,  0,0,
    {7,3,4,  3,4,1,  2,6,3,  4,1,4,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,
    {7,3,4,  1,3,4,  7,2,4,  3,1,4,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,
    {7,3,1,  1,3,4,  3,2,6,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,
    {6,3,1,  3,4,1,  2,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,
    {6,6,4,  1,3,1,  3,6,4,  4,3,3,  1,4,2,  4,6,3,  2,1,4,  5,3,4,  5,3,1,  0,0,
    {7,3,4,  4,2,3,  4,2,6,  1,4,7,  3,1,4,  2,6,2,  3,1,3,  4,1,1,  4,0,0,  0,0,
    {7,3,1,  4,6,3,  3,4,1,  3,4,4,  3,3,4,  4,1,3,  4,4,0,  0,0,0,  0,0,0,  0,0,
    {7,3,4,  3,4,1,  3,4,3,  4,1,3,  4,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,
    {7,3,1,  1,3,4,  1,3,4,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,
    {6,6,1,  3,4,6,  1,3,1,  6,4,3,  3,4,4,  3,2,1,  0,0,0,  0,0,0,  0,0,0,  0,0,
    {6,1,3,  3,4,3,  3,4,3,  1,1,4,  3,4,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,
    {4,4,3,  3,4,3,  4,3,1,  4,3,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,
    {4,1,3,  3,1,2,  3,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,
    {3,4,1,  3,1,1,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,0,  0,0,
};


/***************************************************/
/* images to be compared with the library */
/***************************************************/

double x_image1[11] =          /* MIRACS image # 12 */
     {3,4,3,0,6.5,7.5,7.5,8,12,3,4};
double y_image1[11] =
     {0,0.5,1,1,1,3.2,1,1.0,0,0.5};

double x_image2[10] =          /* arbitrary image */
     {1.4,2.4,2.3,3.2,3.7,4.6};
double y_image2[10] =
     {2.3,3.12,3.8,3.7,2.1,2.6};

main()
{
    int i,code[30],flip_code[30],tmp_code[30],flag;
    code_gen(x_image1,y_image1,code,flip_code,10);
    stdize(code,tmp_code,10);
    stdize(flip_code,tmp_code,10);
```

```
printf("\n code of imagel is :\n");
for(i = 0 ; i < 10 ; ++i)
{
      printf(" %d ",code[i]);
}
printf("\n flip_code of imagel is :\n");
for(i = 0 ; i < 10 ; ++i)
{
      printf(" %d ",flip_code[i]);
}
compare_code(code,lib,&flag,10);
compare_code(flip_code,lib,&flag,10);
code_gen(x_image2,y_image2,code,flip_code,8);
stdize(code,tmp_code,8);
stdize(flip_code,tmp_code,8);
printf("\n code of image2 is :\n");
for(i = 0 ; i < 8 ; ++i)
{
      printf(" %d ",code[i]);
}
printf("\n flip_code of image2 is :\n");
for(i = 0 ; i < 8 ; ++i)
{
      printf(" %d ",flip_code[i]);
}
compare_code(code,lib,&flag,8);
compare_code(flip_code,lib,&flag,8);
if(flag == 0)
{
      printf("\n This object is not recognized \n");
}
}
```

```
code of imagel is :
     4  1  4  6  3  1  5  6
flip_code of imagel is :
 7  1  4  5  3  1  4  6  3  6
This is a 'Mirage' airplane

code of image2 is :
       1  3  3  4  4  3
flip_code of image2 is :
   6  1  1  3  6  4  1
This object is not recognized

code for imagel is :
 0410342261436
 the flip_code for imagel is :
 10416342261436
 this is a 'Mig' airplane

 this is a 'Mig' airplane

 code for image2 is :
 7060413214
 the flip_code for image3 is :
 7060113614
 This is a 'Phantom' airplane

 code for image3 is :
 7111103
 the flip_code for image3 is :
 7113333
 this object is not recognized
```

APPENDIX B

ACTUAL PROFILES OF SIX FIGHTER PLANES AND THEIR POLYGONAL APPROXIMATIONS

USED TO TEST THE 3-D INSM ALGORITHM

$$\frac{28}{77} = 36\% \text{ USED}$$

Figure 14. One quadrant of the mirage projection library.

$$\frac{24}{77} = 31\% \text{ USED}$$

Figure 15. One quadrant of the mirage projection library.

$$\frac{29}{77} = 38\%$$ USED

Figure 16. One quadrant of the phantom projection library.

$$\frac{30}{77} = 39\%$$ USED

Figure 17. One quadrant of the F 104 projection library.

$$\frac{29}{77} = 38\% \text{ USED}$$

Figure 13. One quadrant of the B 57 projection library.

$$\frac{29}{77} = 38\% \text{ USED}$$

Figure 18. One quadrant of the F 105 projection library.

Figure 20. Polygonal approximations of the mirage profiles.
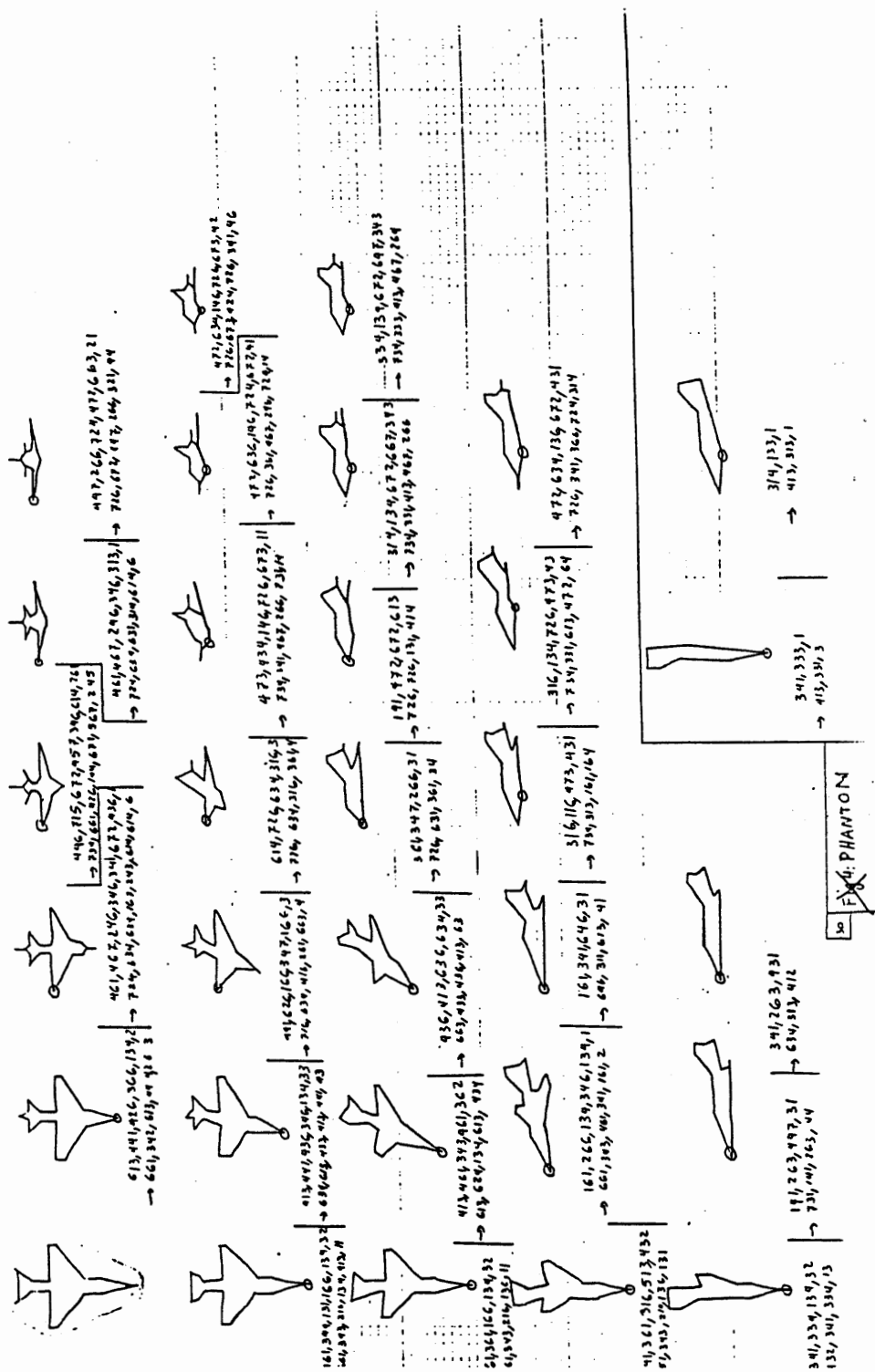
Figure 21. Polygonal approximations of the mig profiles.

55

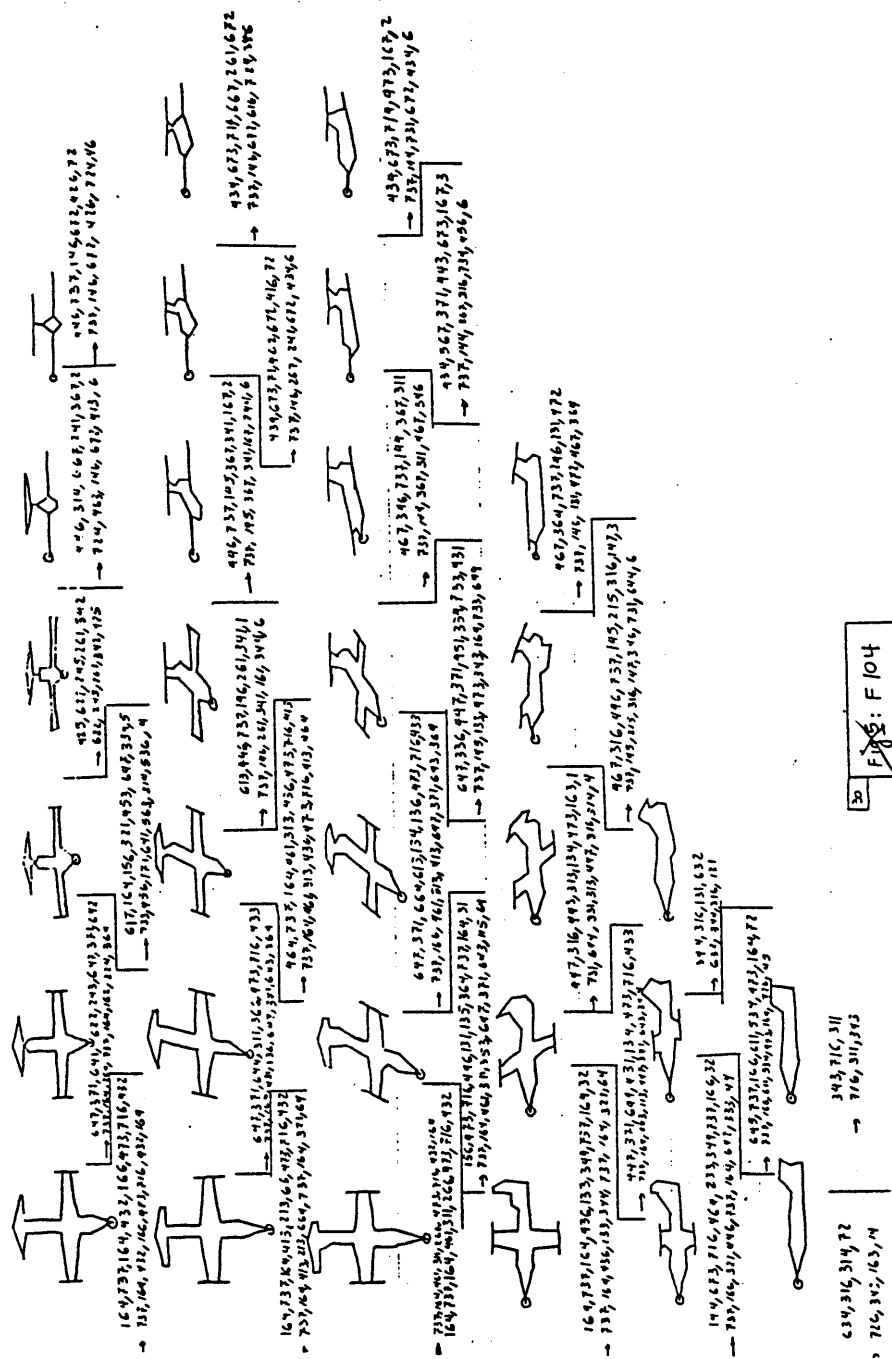Figure 22. Polygonal approximations of the phantom profiles.

Figure 23. Polygonal approximations of the F 104 profiles.
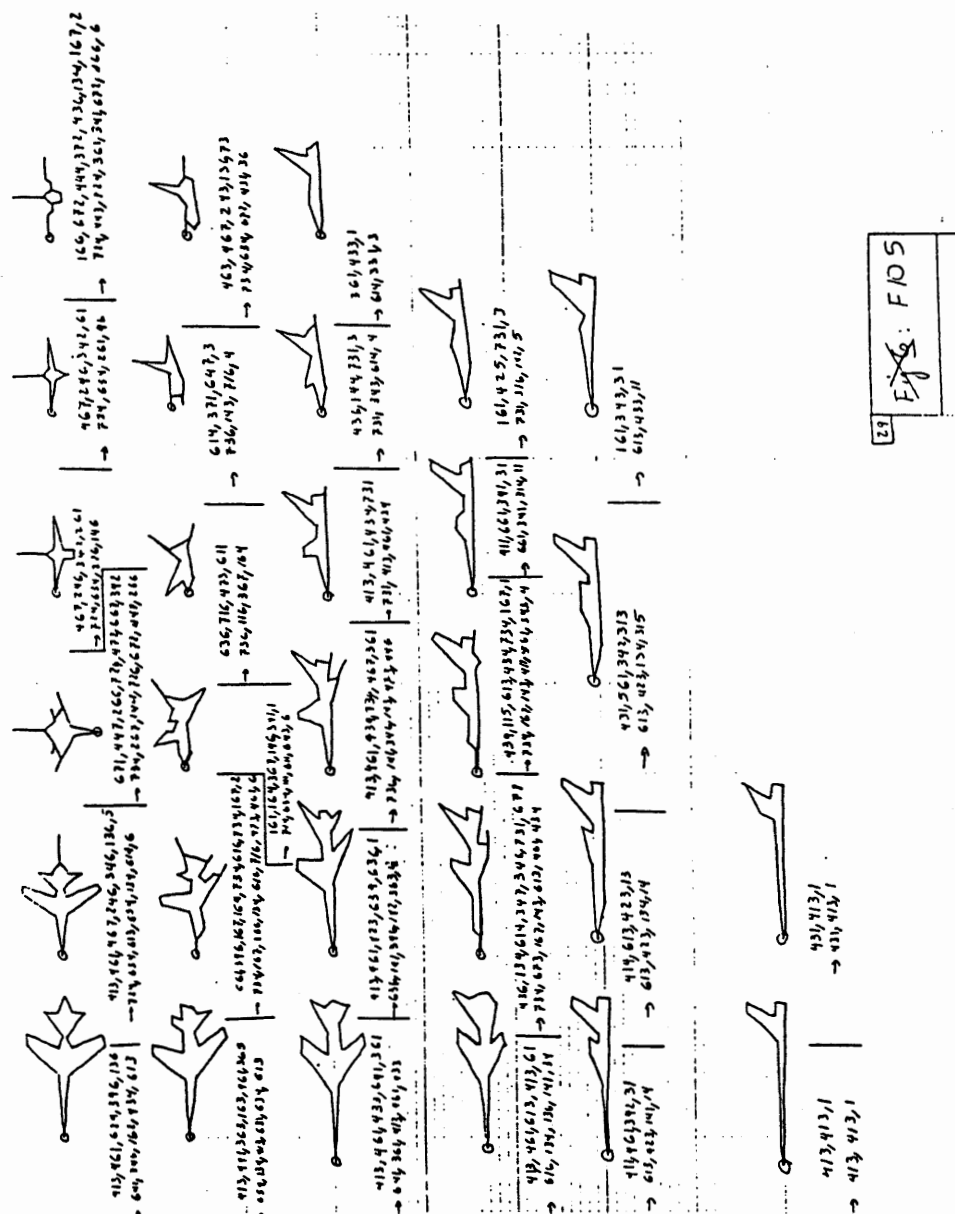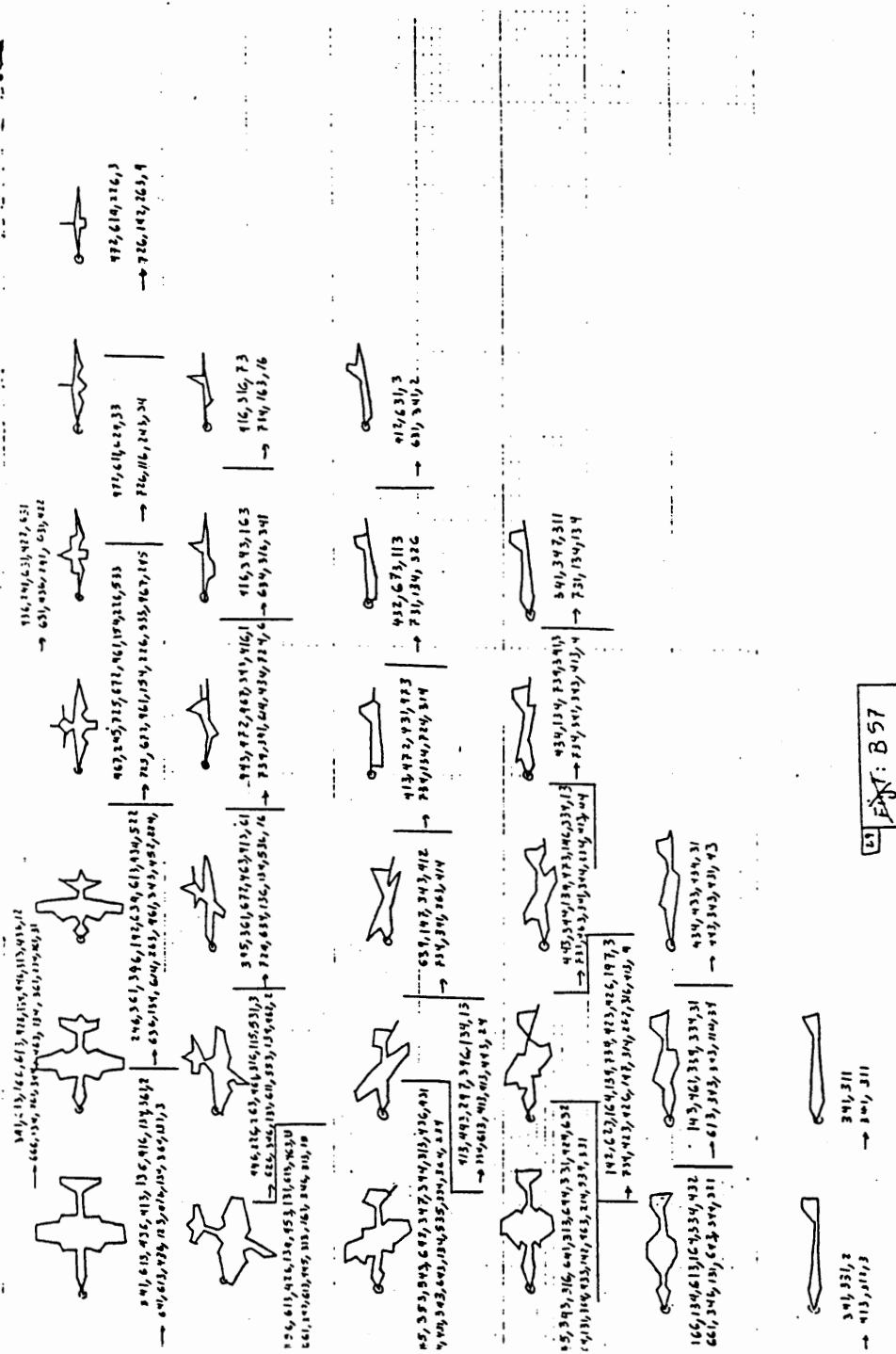
Figure 24. Polygonal approximations of the F 105 profiles.

Figure 25. Polygonal approximations of the B 57 profiles.