

# **ON OBJECT REPRESENTATION SCHEMES**

THESIS SUBMITTED IN PARTIAL FULFILMENT OF THE  
REQUIREMENT FOR THE AWARD OF THE DEGREE OF  
**DOCTOR OF PHILOSOPHY**

By  
**BOGGAVARAPU LAVAKUSHA**



SCHOOL OF MATHEMATICS & COMPUTER/INFORMATION SCIENCES  
**UNIVERSITY OF HYDERABAD**  
HYDERABAD-500 134  
INDIA

**July, 1989**

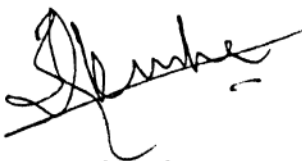
Since GOD could not be everywhere, he created Parents.

to my parents

## STATEMENT

I hereby declare that the matter embodied in this thesis is the result of investigations carried out by me in the School of Mathematics and Computer/Information Science, University of Hyderabad, Hyderabad, India under the supervision of Prof. P.G. Reddy and Dr. A.K. Pujari.

In keeping with the general practice of reporting scientific observations due acknowledgements have been made wherever the work described is based on the findings of other investigators.

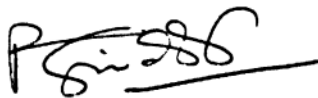


Lavakusha B.

Hyderabad, July 24<sup>th</sup>, 1989.

# C E R T I F I C A T E

Certified that the work contained in this thesis entitled:  
"ON OBJECT REPRESENTATION SCHEMES" has been carried out by  
Boggavarapu Lavakusha, under our supervision and the same has not  
been submitted elsewhere for any degree.

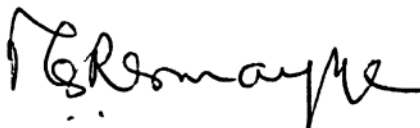


Prof. P.G. Reddy.

  
25/07/89

Dr. A.K. Fujari.

(Doctoral Supervisors)



DEAN

DEAN  
School of Mathematics and  
Computer/Information Science  
University of Hyderabad  
Central University P O  
HYDERABAD - 500 134

25/7/89  
School of Mathematics & Computer/Information Science.



## A C K N O W L E D G E M E N T S

It is always a pleasant task to acknowledge the role of those who are responsible for me being what I'm today.

At the outset, I love to express my gratitude to my doctoral supervisors, Prof. P.G. Reddy and Dr. A.K. Pujari, who had made my long cherished dream come true. I thank them for their constant support, encouragement and inspiration.

I also thank Prof. K. Viswanath, supervisor for my M.Phil project, for having taught me the art of analysis.

I thank Prof. M. Sitaramaiah, the Dean, School of Mathematics and Computer/Information Science, and the previous Deans, for providing the necessary facilities in the school. And Dr. Malay Dutta, Head, Department of Computer Science, Gauhati University, for creating an atmosphere where I could work freely.

I am thankful to all my teachers in Sainik School, Korukonda, P.R.Government College, Kakinada, and the University of Hyderabad, from whom I have learnt many things, besides academic issues.

I am grateful to Dr. B.E. Prasad for introducing me to this fascinating and exciting field of computers.

I am grateful to the men and machines of the computer centers at Indian Statistical Institute, Calcutta, Indian Institute of Technology, Kanpur, University of Hyderabad, Hyderabad, and Gauhati University, Guwahati where I'd befriended the 'intelligent' machines.

I acknowledge the numerous discussions, both academic and otherwise, that I had with my colleagues, Uma in particular, in the school. I thank them.

Ayn Rand and Swami Vivekananda have inspired me and made me realize my SELF. I am indebted to them.

I have the nasty habit of falling into soups and it was friends like Prabhakar, Ramji, Sastry and Venu who fished me out. I owe them a lot.

It was friends Ramesh, Vishwanath, Sekhars (B.S and G.S), Suri, Raghu, Chandu, Muni, Vasu, Mr. Kishan (now Dr.) who made the life here a pleasant one. I am oblized to them.

Dr. Pavan Kumar (then Mr. ) supplied the necessary zeal and fire to carry on. I owe him ~~a lot~~ two lots.

Mrs. Q.T. Chaliha did a meticulous job of converting a manuscript into an ASCII file. I thank her. I am oblized to Sandilya for having gone through earlier versions of the thesis very painstakingly and for his assistance.

I thank my brothers for their support - both financial and moral. I am grateful to the UGC for the financial assistance during part of my doctoral studies.

## CONTENTS

|             |  |    |
|-------------|--|----|
| 1           | Introduction                                       | 1  |
| 2           | 2D and 3D object representation schemes : A survey | 8  |
| 2.1         | 2D object representation                           | 10 |
| 2.1.0       | Point data   | 10 |
| 2.1.0.1     | Point Quadtrees                                    | 11 |
| 2.1.0.2     | k-d Trees  | 14 |
| 2.1.1       | Curvilinear Data                                   | 15 |
| 2.1.1.1     | Polylines  | 16 |
| 2.1.1.2     | Chain Codes  | 18 |
| 2.1.1.3     | Strip Trees  | 21 |
| 2.1.1.4     | Edge Quadtrees                                     | 23 |
| 2.1.2       | Region Representation                              | 25 |
| 2.1.2.1     | Spatial Occupancy Arrays                           | 26 |
| 2.1.2.2     | Y-Axis   | 26 |
| 2.1.2.3     | Bintrees   | 29 |
| 2.1.2.4     | Pyramids   | 31 |
| 2.1.2.5     | Region Quadtrees                                   | 32 |
| 2.1.2.5.1   | Pointered Quadtrees                                | 33 |
| 2.1.2.5.2   | Pointerless Quadtrees                              | 49 |
| 2.1.2.5.2.1 | DF-expressions                                     | 49 |
| 2.1.2.5.2.2 | Linear Quadtrees                                   | 50 |
| 2.1.2.5.3   | Semi pointered trees                               | 58 |
| 2.1.2.5.3.1 | One-of-Four Quadtrees                              | 58 |
| 2.1.2.5.3.2 | Goblin Quadtrees                                   | 60 |
| 2.1.2.6     | Skeletons and Medial Axis Transform                | 62 |

|     |         |                                       |     |
|-----|---------|---------------------------------------|-----|
|     | 2.1.2.7 | QMAT                                  | 65  |
|     | 2.1.2.8 | TID                                   | 66  |
| 2.2 |         | 3D Object Representation              | 69  |
|     | 2.2.1   | Spatial Enumeration                   | 70  |
|     | 2.2.1.1 | Spatial Occupancy Arrays              | 70  |
|     | 2.2.1.2 | MAT                                   | 71  |
|     | 2.2.1.3 | Octrees                               | 72  |
|     | 2.2.2   | Cell Representation                   | 79  |
|     | 2.2.2.1 | Rectangular Parallelepiped            | 79  |
|     | 2.2.2.2 | Translation Invariant Data structures | 84  |
|     | 2.2.3   | Constructive Solid Geometry           | 86  |
|     | 2.2.4   | Generalized Cylinders                 | 88  |
|     | 2.2.5   | Boundary Representation               | 90  |
|     | 2.2.5.1 | Space Curves                          | 90  |
|     | 2.2.5.2 | Surfaces                              | 91  |
|     | 2.3     | Conclusions                           | 92  |
| 3   |         | Edge k-d Trees                        | 95  |
|     | 3.1     | Edge k-d Trees                        | 95  |
|     | 3.2     | Construction of the Tree              | 96  |
|     | 3.3     | The Algorithm                         | 101 |
|     | 3.4     | Analysis                              | 104 |
|     | 3.5     | Operations on Edge k-d Trees          | 105 |
|     | 3.5.1   | Point Membership                      | 106 |
|     | 3.5.2   | Scaling                               | 108 |
|     | 3.5.3   | Translation                           | 109 |
|     | 3.6     | Conclusions                           | 110 |
| 4   |         | Linear Octrees by Volume Intersection | 112 |
|     | 4.1     | Volume Intersection                   | 113 |

|       |   |     |
|-------|---|-----|
| 4.2   | Linear Octrees  | 119 |
| 4.3   | Linear Octrees through Volume Intersection                        | 122 |
| 4.3.1 | The Algorithm   | 123 |
| 4.3.2 | Analysis  | 128 |
| 4.4   | Octrees through Volume Intersection                               | 129 |
| 4.5   | Conclusions   | 133 |
| 5     | Volume Intersection Algorithm with<br>Changing Direction of Views | 136 |
| 5.1   | Problem Specification   | 144 |
| 5.2   | The Algorithm   | 145 |
| 5.2.1 | Updating The VI-hull  | 146 |
| 5.2.2 | Fixing a Voxel  | 148 |
| 5.3   | Accuracy Measures   | 153 |
| 5.3.1 | Determination of the New Direction                                | 155 |
| 5.4   | Generation of Linear Octree                                       | 156 |
| 5.5   | Conclusions   | 156 |
| 6     | Conclusions   | 158 |
|       | References  | 163 |

---

# CHAPTER 1

## Introduction

---

A tremendous progress has been made in the field of computers since the day von Neumann developed the first electronic computer, ENIAC, in '45. In the early days, the computers were used for mere numerical calculations. But now, they can process a variety of information - numeric, character (non-numeric), symbolic, pictorial and even knowledge. That the present day computers possess the ability to process knowledge, is what makes it interesting in a variety of applications. The purpose of Artificial Intelligence (AI) is to incorporate some intelligence into hitherto 'not-so-intelligent' machines. The acquisition, representation and processing of knowledge are the major branches of AI.

But, some of the tasks which are very easy for the humans are extremely difficult and in some cases impossible for the machine to perform. Humans recognize the faces and objects very easily compared to machines. These are applications where commonsense, a character of human beings, is required to solve problems and the quantification of knowledge into the machines will take a long time for researchers. The lack of a thorough understanding of the underlying mechanism of the thinking and other human actions related to human brain is the major bottleneck for further development in the field of AI in computer science.

The goal of the Fifth Generation Computers is to make the machine as intelligent as the human being, if not, better. Computer Vision, besides AI and Natural Language Processing (NLP), play a crucial role in the development of the 'friendly and intelligent machine'.

It is not easy to draw the lines of demarcation between Computer Vision, Computer Graphics and Image Processing. Image processing deals with acquisition and preprocessing of pictorial information like photographs, TV camera film, satellite images, aerial photographs, hand-written characters, typed documents, signatures, fingerprints and buildings' blueprints.

The display of objects is dealt by computer graphics i.e., to display the objects on the monitor in as realistic a form as possible. In fact, the display of 3D objects on the 2D monitor is difficult because of the disparity in the dimensions involved. To make better the display of certain straight lines and curves distorted due to the pixel resolution, is also the concern of computer graphics. Some other aspects it deals with are windowing and clipping, elimination of hidden lines and surfaces, shading and user friendly interaction.

Computer Vision (CV) is the study of representation, retrieval (in part or full) and processing of pictorial information. Processing may mean either image understanding, or the search for or location of a particular object in a satellite image or an aerial photograph, identification of 3D objects in a 2D image of a 3D scene, reconstruction of the original 3D object model from its 2D slices (as in CAT) or provide the necessary path for a mobile robot to 'walk' from one corner of the room to

another corner, without colliding with any objects.

The representation of information, be it pictorial or knowledge, plays a vital role, because all the succeeding operations depend heavily on the representation model. One of the major challenges in computer vision is to represent objects or the important aspects of it, so that they may be learned, recollected, matched against and used. Several factors hamper the study of object representation.

Most of the advances in AI research made so far relates to the knowledge about the imitation of the working of the human brain. Since the human brain functions like 'seeing' and 'thinking' are difficult to quantify, the imitation technique cannot be carried over easily to the respective fields of study, viz. Computer Vision and AI. In the human vision, although the image that falls on the eye's retina is two dimensional, yet the human eye can 'grasp' the 3D scene, possibly because of the binocular vision, vergence, accommodation, relative motion, occlusion, shading and ,structure and size. If only one can use these cues in computer vision, then the machines can perform the function of 'seeing'.

The complexity of objects, in terms of its shape, could be an obstacle. The color, motion and intensity of an object can be quantified, but not its shape!. Certain simple objects can be broken up into the primitive blocks : regular geometrical objects like cylinders, cones, spheres etc. and can be represented as a combination of these primitives. But, this system too, has got some drawbacks.



Albeit, object recognition is very normal and casual for humans, object description is difficult to quantify. An attempt to describe a human face in words, even with the help of fuzzy set theory will be futile. When it comes to the machines, the objects are to be represented in the memory for further operations and/or actions. The efficiency of the machine to act, for example, for a robot to move from point A to point B, depends on the representation of the information in the memory. Further, the representation of 2-dimensional (2D) and 3-dimensional (3D) objects is essential for computer graphics, image understanding, CAD/CAM, Geographic Information Systems (GIS), solid modeling, and other related areas.

The lack of a precise language for object description is the reason for the inaccessibility of the object-processing algorithms and data structures. Many data structures for object representation are in use for storage, retrieval, modification and efficient query answering. These data structures have been designed to represent either a particular structural relationship, to save space or to allow for fast access of data. The wide spectrum of representation schemes can be classified based either on the underlying data structure, chronology (development or on a particular property (translation invariance, fast query processing etc.)). However one cannot find any one single representation scheme for 2D and 3D objects that best suits all the requirements of the user community.

The present work concerns with some of the representation schemes of 2D and 3D objects. In the first part, a new representation scheme for storing polygons in 2D is proposed.

In a GIS, a query that is very frequently asked relates to the "point-in-polygon", i.e., given a polygon  $P$  and a point  $p$ , to determine whether the point is inside the polygon. This problem is also vital for path planning of a mobile robot. A representation scheme based on balanced binary tree, called the edge  $k$ -dimensional balanced binary search tree or edge  $k$ -d tree for short is presented here. The edge  $k$ -d tree is constructed by recursively dividing the set of vertices by vertical and horizontal cuts along the median vertex sorted on  $X$  and  $Y$  axes respectively. The proposed data structure has the following important features :

- i) The region is represented as balanced binary tree.
- ii) The tree structure does not change when the polygon is translated or scaled.

Thus the edge  $k$ -d tree representation facilitates efficient method for point-in-polygon, translation and scaling operations. So it can be the most suitable data structure for region representation in situations like automated cartography, GIS, and Robotics where these operations are required very frequently.

In the second part, we concentrate on getting the desired 3D object representation from 2D projections. As a majority of the image acquisition systems are capable of capturing 2D data it is easier to acquire areal information than that of volumetric information. An efficient method to generate the 3D object representation is through capturing a set of its 2D silhouettes and intersecting the corresponding swept volumes. When a silhouette of an object is swept along the viewing direction in

the 3D space, it generates a cylinder (also called swept volume). The cylinders, so obtained by sweeping several silhouettes are intersected to obtain a representation of the original object. Using this technique of volume intersection, a method for generating the linear octree of the 3D object is proposed. The volume intersection method generates an approximation of the original object from a set of 2D silhouettes. There is no definite rule to decide the directions in which the silhouettes should be taken to get the best possible approximation. In this work the problem of determining additional viewing directions so as to refine the approximated construction is also addressed.

The selection of the proper representation scheme is crucial and, ironically is difficult too, when there is an abundance of the representation schemes. To make the selection easy, a review of the major representation schemes is presented in Chapter 2. Though there are some reviews of the object representation techniques earlier, (cf. Samet [84a], Shapiro [79] and Srihari [81],) none of them cover the whole spectrum and moreover there have been continuous additions. The representation schemes for 2D and 3D objects is presented separately. The classification of the schemes is based on the object since in a natural sequence the object comes first, followed by the representation. Further, the classification is based on the dimensionality of the components in a 2D object, namely points, curves and regions. The different operations that can be performed and queries that can be asked of are also presented.

A method to represent 2D polygons is proposed in Chapter 3, viz. the edge  $k$ -dimensional balanced binary search tree (edge  $k$ -d

tree, for short). Some of the algorithms for answering the point-in-polygon query, and for operations of translation and scaling are presented. The edge k-d tree for a polygon of  $N$  vertices can be constructed in time proportional to  $O(N \log^2 N)$ .

Chapter 4 concerns with the volume intersection. A method to generate the linear octree of the 3D object using the front, the side and the top 2D silhouettes of the 3D object is proposed. The 2D silhouettes are given in the form of raster images. The linear octree is built in time of  $O(N^2 + R)$ , where  $N$  is the size of the universe and  $R$  is the number of nodes in the linear octree. We show that our method takes less time than an existing method which uses quadrees and octrees for representing the 2D silhouettes and the resulting 3D object respectively.

We imbibe some learning concepts into the volume intersection in Chapter 5. One of the drawbacks of the volume intersection is that it 'fills' up certain concavities in 3D objects. This can be partially overcome by considering more than three views. A scheme to measure the degree of approximation is proposed which takes into account the ratio of the part of the volume that is unambiguous to that of the ambiguous one. To minimize the approximation error, a heuristic that suggests a new viewing direction, from which the next silhouette of the object is to be taken is presented. By doing so, the approximation is refined gradually.

In Chapter 6, the overall conclusions of our study is presented. We also present certain unresolved problems. A few suggestions are made for further work.

---

## CHAPTER 2

### 2D and 3D object representation schemes : A survey

---

Several data structures and representation schemes exist for the storage of 2D and 3D objects. Selecting the proper representation scheme for input data, intermediate structures and output data is a crucial factor for the construction of a good algorithm. A data structure may be chosen to represent a particular structural relationship, to save space or to allow for fast access of data. Different data structures have been developed to cater to the above requirements by many researchers. But no single data structure has all the desired properties. Hence, as a result, some more representation schemes have been developed. Depending on the need, one chooses the representation scheme that suits best. In view of the plethora of these representation schemes, to make the selection easy, a review of the important schemes of representation of 2D and 3D objects in the context of their construction, various operations that can be performed and their space and time complexities, is presented in this chapter. In section 2.1, the representation schemes for 2D objects is reviewed and in section 2.2, for 3D objects.

A few of the queries that could be addressed to any representation scheme are

- (i) Point-membership: given an object representation and a point, to determine if the point lies inside the object.
- (ii) Computation of the area (volume), perimeter (surface area)

for 2D (3D) objects.

- (iii) The primitive set operations of intersection and union of two objects, A and B, complement of an object and the set difference of A and B. Using these primitive operations, many more complex image processing tasks can be performed.
- (iv) Geometric transformations, namely, translation, rotation and scaling. Any geometric transformation can be obtained by a composition of these three operations.
- (v) Connected component labeling: A connected component may be defined as a region in which any pair of points may be connected by a curve lying entirely inside the region. It is also desirable to count the connected components.
- (vi) Euler number: Also called Genus, it may be defined as the number of holes subtracted from the number of connected components for a region. For a planar graph, it is defined as

$$G = V - E + F,$$

where V, E and F are the number of vertices, edges and faces, respectively. But, for a binary image these can be interpreted as [Minsky et al 69]

V = number of black pixels

E = number of horizontal or vertical adjacent  
pairs of black pixels.

F = number of 2 x 2 blocks of black pixels.

Now let us look into various data structures for the storage of objects.

## 2.1 2D OBJECT REPRESENTATION

In this section, by an object, we mean a 2D object. The classification of various data structures, paradigms and representation schemes used for object representation is not simple because the dividing lines are not well drawn. The classification can be made depending on the object we would like to store or on the framework of the data structures used. In the second mode if the classification is based on the underlying approaches, this may be done depending on tree and non-tree structures or on those that are translation invariant and those that are not. But we follow the first mode, i.e., the classification based on the object because in a natural sequence the object comes first followed by the representation. A 2D-object can be visualized as having components of smaller dimensions. We classify the representation schemes based on the dimensionality of these components.

The classification is based on representations that store:

- (i) point data, ( 0-dimension)
- (ii) curvilinear data, (1-dimension) and
- (iii) region data. (2-dimension)

### 2.1.0 POINT DATA

The storage, retrieval and ability to alter point data in a 2D region is very frequently used in geographic information systems (GIS). In a map city headquarters, administrative centres and important places are represented by points with appropriate coordinates. A sample query related to such a point data might be

"Find all the cities which are within 250 km of New Delhi and south of Dehradun".

Storing only the coordinates of cities is not desirable because this does not allow for an efficient query answering.

In the following subsections some of the data structures that store point data in a 2D image are presented. In this subsection the different methods of representation of point data are discussed.

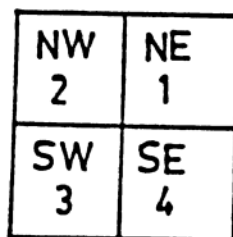
#### 2.1.0.1 POINT QUADTREES

The term point quadtree is employed to denote the quadtree as defined by Finkel and Bentley [74] and to avoid confusion with other types of quadtrees, which are presented in following sections (the history of quadtrees is also deferred).

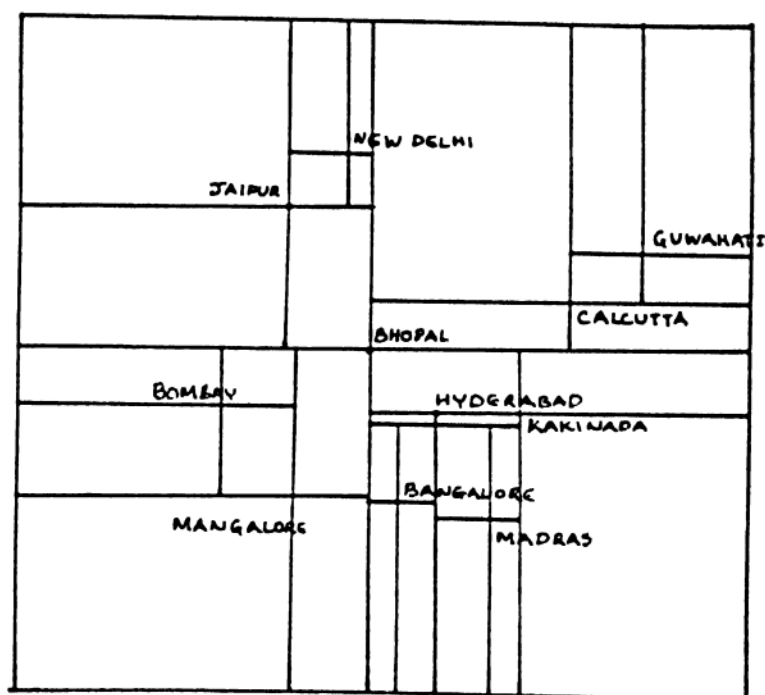
The Point Quadtree, a tree with outdegree 4, is a divide-and-conquer paradigm that divides the image into four quadrants. In the point quadtree, the location of records with 2D keys is stored in the nodes. Each node stores one record and may have at most four sons. The root of the tree corresponds to the entire universe and its four sons divide the universe into four quadrants, viz. NE, NW, SW, and SE with the obvious meaning. Here it is assumed that the NE and SW sons are closed and the NW and SE are open (in the sense of point-set topology).

In a point quadtree, inserting a new record  $(x,y)$  is trivial, and can be done in  $O(\log n)$  time, where  $n$  is the number of records [Finkel and Bentley 74]. At each node, a comparison is made and the correct subtree is chosen for the next test; upon falling out of the tree, the record is inserted at the proper

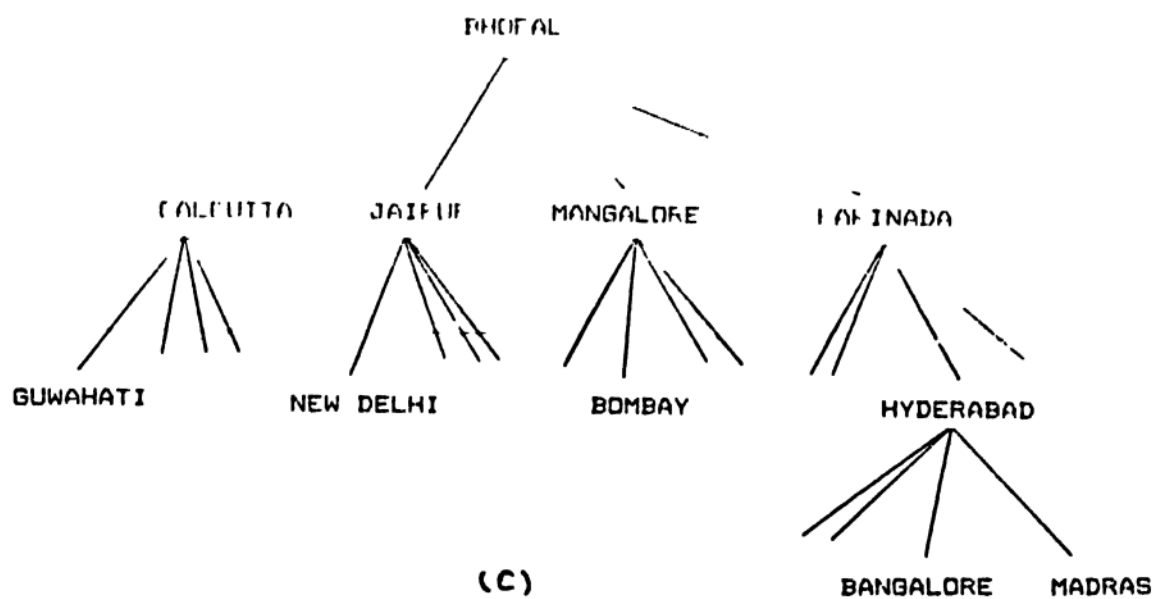




(a)



(b)



(c)

Fig. 2.1 : A point quadtree and the records it represents (a) Numbering for the sons, (b) cities (not to scale) and (c) the representation.

place. For instance, if the record  $(x,y)$  is to be inserted and coordinates of the record stored in the leaf node are  $(p,q)$ , then the record with  $(x,y)$  will be

NE son if  $x \geq p$  and  $y \geq q$

NW son if  $x < p$  and  $y > q$

SW son if  $x \leq p$  and  $y \leq q$

SE son if  $x > p$  and  $y < q$

Search queries are of two types - point search and region search. A point search could be like "Determine the address of the node denoting the point  $(94,61)$ , if it exists". The second, a region search, is invoked by "What are all the cities, within a circle of radius 270 km, centered at  $(77,62)$  ?". The point search basically is a tree traversal and can be accomplished in  $O(\log n)$  time, if there are  $n$  records. However range search takes  $O(n \log n)$  time.

The deletion of records from point quadrees turns out to be very difficult to perform. The difficulty lies in deciding what is to be done with the subtrees attached to the deleted node. They are to be merged with the rest of the tree, which is not an easy process. The only way out, is to reinsert all of the stranded nodes, one by one, into the new tree. This operation of deleting a node is of  $O(n \log n)$ , where  $n$  is the total number of nodes in the two trees to be merged.

The point quadtree is an efficient means of storage for 2D data, when the data is not very dynamic and searches are high. The basic concepts involved can easily be generalized to an arbitrary number of dimensions. The k-d tree is a step in this regard.

### 2.1.0.2 k-d TREES

The k-dimensional binary search tree [ Bentley 75 ], k-d tree for short (this terminology is due to Donald.E.Knuth), is devised to organize a set of points in multidimensional space. In a 2D space, it becomes a 2-d tree. For the construction of a 2-d tree, the map space is recursively partitioned into rectangular blocks by a set of straight lines parallel to the axes. The partitioning process for the generation of 2-d tree is given below.

- (a) The lengths of the horizontal and vertical sides of the rectangle to be partitioned are computed.
- (b) Divide the rectangle into two by a straight line parallel to the shorter side. The position of the dividing line is determined so that each of the resultant smaller rectangles contain the same amount of data.
- (c) To each resultant rectangle, (a) and (b) is recursively applied until the amount of data in any rectangle becomes "manageable".

The manageable amount of data depends on the application. Matsuyama et al [84] used the 2-d tree for storing data in a geographic information system (GIS), where they recursively divide the rectangles till the amount of data in any rectangle becomes less than the capacity of a memory page. An example of partitioning by 2-d tree where the capacity of a page is 2 is given in Fig.2.2

Here, a node in the 2-d tree represents a rectangular region generated by the partitioning process and contains the

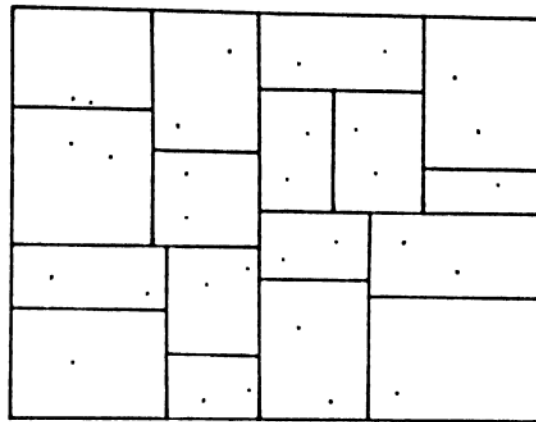


Fig. 2.2 : A k-d tree with a page capacity of 2.

position and the direction of the dividing line of the region. The root node denotes the whole map space, and, the leaves, the ultimate blocks. At the beginning, the tree is balanced, but as many records are inserted/deleted, the tree may become heavily unbalanced and may necessitate the reorganization. Friedman et al [77] have given an algorithm for finding best matches in  $O(\log n)$  time, if there are  $n$  records. Murphy and Selkow [86] used the k-d tree for efficiently retrieving from a file of fixed length binary key words the best match (in Hamming metric) to a given input word.

### 2.1.1 CURVILINEAR DATA

In this category, the boundary of the object is approximated by straight line segments and information relating to these segments is stored. Owing to inherent nature of the representational approach, computation of perimeter and performing geometric operations are done easily.

### 2.1.1.1 POLYLINES

A polygon is a closed polyline of a finite number of line segments. A polygon,  $P$ , of  $N$  edges can be represented either by the equations of the straight lines or equivalently, by the coordinates of the vertices. Usually, the vertices in a cyclic order and the space requirement for storing a polygon of  $N$  sides by the vertices is of  $2N$ , whereas by the line equations it is  $3N$ . The vertices can be stored by means of an array or a circular linked list. This scheme of representation is also referred to as vector representation in the literature.

The computation of perimeter is by summing over the lengths of the line segments, joining the vertices,  $(x_i, y_i)$ ,  $i = 0, 1, \dots, N-1$ .

$$\sum_{i=0}^{N-1} [(x_i - x_{i+1})^2 + (y_i - y_{i+1})^2]^{1/2}$$

Here subscript calculations are modulo  $n$ .

The area  $\Delta$  is given by

$$\Delta = 1/2 \sum (x_{i+1}y_i - x_iy_{i+1}).$$

Since the three basic geometric transformations are homomorphic, i.e., for a transformation,

$$T : R^2 \longrightarrow R^2$$

$$T(X+Y) = T(X) + T(Y), \quad X, Y \in R^2,$$

the transformations can be applied on the vertices and the transformed vertices can be joined to obtain the transformed polygon. The homogeneous coordinates [Newman et al 84] are used to represent transformations by matrices. For this, the dummy variable,  $w$ , is made use of. In the new coordinate system  $(x, y)$  becomes  $(xw, yw, w)$ . For our purposes,  $w$  is assigned 1. Hence a

translation by  $(T_x, T_y)$ , in the new system would be

$$(x \ y \ 1) \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ T_x & T_y & 1 \end{pmatrix} = (x+T_x, y+T_y, 1).$$

Rotation of  $\theta$  degrees about the origin,

$$(x, y) \rightarrow (x \cos \theta + y \sin \theta, -x \sin \theta + y \cos \theta)$$

is given by

$$(x \ y \ 1) \begin{pmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{pmatrix} = (x \cos \theta + y \sin \theta, -x \sin \theta + y \cos \theta, 1).$$

Scaling, equivalently called zooming, is the change in size and proportion of the polygon. For the scaling by factor  $s$ ,

$$(x, y) \rightarrow (x S_x, y S_y),$$

the matrix form is

$$(x \ y \ 1) \begin{pmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{pmatrix} = (x S_x, y S_y, 1)$$

When  $S_x = S_y$ , the polygon is enlarged or reduced depending on whether  $S_x$  is greater or less than 1. If  $S_x$  and  $S_y$  are different, the scaling has the effect of distorting polygons by elongating or shrinking them along directions parallel to the coordinate axes. The mirror images can be obtained by assigning negative values to  $S_x$  or  $S_y$ .

The membership problem of a point  $p$  in a simple  $n$ -gon,  $P$ , can be answered in  $O(n)$  time, without any preprocessing [Preperata et al 85]. The algorithm is based on the Jordan Curve theorem. Consider the horizontal line  $l$  passing through  $p$ . Assuming that  $l$  does not pass through any vertices of  $P$ , if  $L$ , the number of intersection of  $l$  with the boundary of  $P$  to the left of  $p$ , is odd, the point is internal. When  $l$  passes through

one of the vertices of  $P$ , a small rotation of  $l$  around  $p$  removes the degeneracy.

```

-----
function MEMBER (p,P)
begin
  L=0
  for i = 1 to N do
    begin
      if (edge (i) is not horizontal) then
        if (lower extreme of edge (i) intersects to the left of p)
          then L=L+1
    end
  if (L is odd) then p is internal else p is external.
end.
-----

```

#### Algorithm 2.1

Using the vector representation, one can reconstruct the polygon exactly, but the display on the monitor is effected by the pixel resolution of the graphic monitor. Besides, this representation is dynamic, i.e., vertices can be added, deleted or modified easily by adding, deleting or updating the corresponding nodes in the linked list.

#### 2.1.1.2 CHAIN CODES

The chain code representation [Freeman 74] is a very commonly used scheme in cartographic applications. This is also known as the boundary or border code.

The Chain codes consist of line segments, one unit long, that lie on a fixed grid with a fixed set of possible orientations. Only the starting point is represented by its location. The coordinates of other boundary points are computed relative to this point. The chain code is a sequence of unit vectors (i.e., one pixel wide) in the four principal directions. The directions are represented by numbers: for example, let the

integer  $i$ ,  $0 \leq i \leq 3$ , represent the direction of 90 degrees. By traversing along the boundary from starting point in clockwise direction, the chain code is generated.

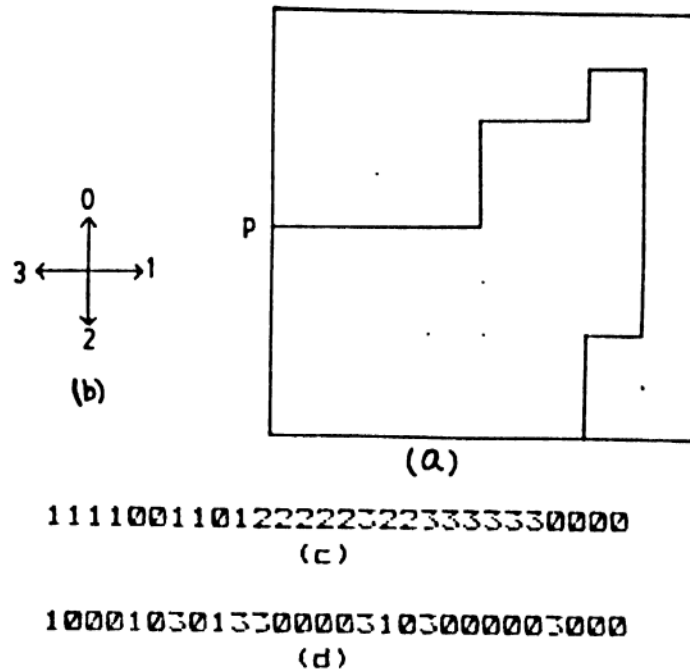


Fig. 2.3 : An object with the grid (a), the directional codes (b), the chain code (c), and its derivative (d).

This is generally written as  $1^4 0^2 1^2 0 12^5 32^2 3^6 0^4$ . A power of  $i$  to the directional code indicates that directional code is repeated  $i$  times. For a better approximation of the curve instead of four directions only, one may choose six or eight or even more number of directions.

The Chain codes facilitate trivial computation of the perimeter by adding up all the powers. Using  $(x,y)$  as the starting point on the boundary, the area can be computed using the function AREA( ).



```
.....
AREA(x,y,chain-code)
```

```
begin
```

```
    area=0
```

```
    ypos=y
```

```
    for each element of chain-code
```

```
        case (element-direction)
```

```
            0 : area=area-ypos
```

```
            1 : ypos=ypos+1
```

```
            2 : area=area+ypos
```

```
            3 : ypos=ypos-1
```

```
end.
```

### Algorithm 2.2

The chain codes allow for representation of planar curves by a simple scheme with minimal storage requirements. Besides, the chain codes are often used in place of original data. The operations of rotation, computation of width and height, testing for symmetry about an axis and the determination of area enclosed or under a chain code can be performed on them [Shapiro 79]. The derivative of a chain code is defined as the sequence of numbers indicating the relative direction of chain code segments; the number of left hand turns of  $\pi/4$  needed to reach the next segment. The chain code and its derivative are given in Fig. 2.3. The derivative is useful because of its invariance to boundary rotation.

They also simplify the detection of features of a region boundary, such as corners [Freeman and Davis 77] or concavities. On the other hand, they do not facilitate the determination of properties like elongatedness and process of set operations of intersection and union [Samet 84a]. A chain correlation function can be defined to give a measure of similarity between two curves. Since the coordinates of the boundary points are stored relative to the starting point, translation is trivial. It involves shifting of the starting point. Scaling by powers of 2

and rotations by multiples of 90 degrees are easy to perform.

### 2.1.1.3 STRIP TREES

The strip tree [Ballard 81] is a hierarchical representation of curves. It is constructed by successively approximating segments of the curve by the smallest enclosing rectangles. The root of the strip tree, a binary tree, represents the entire curve. The node structure consists of eight fields, wherein the first six fields define the rectangle and the remaining two are pointers for the two sons. The original curve is broken up into strips and each strip is defined by a six-tuple  $(x_1, x_2, y_1, y_2, w_l, w_r)$  as shown Fig.2.4

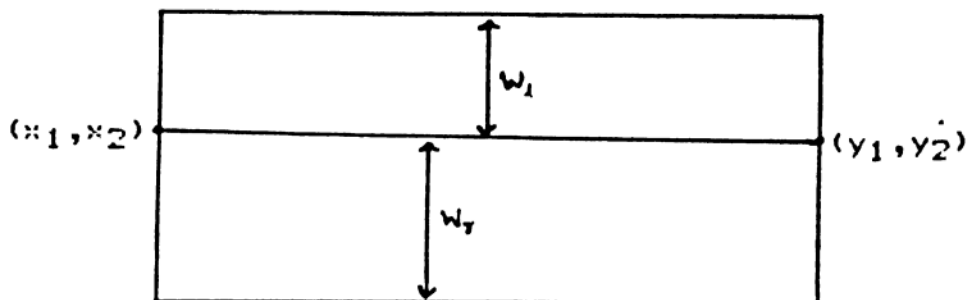


Fig. 2.4 : The six-tuple  $(x_1, x_2, y_1, y_2, w_l, w_r)$  that defines the rectangle to store a strip.

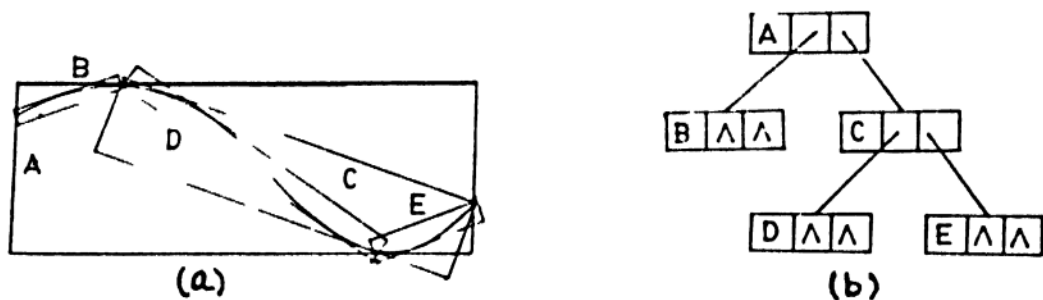


Fig. 2.5 : A curve with the enclosing rectangles (a) and its strip tree (b).

Here  $X=(x_1,x_2)$ ,  $Y=(y_1,y_2)$  are the two end points of the curve and  $w_l$  and  $w_r$  help in defining the rectangle. The algorithm for constructing a strip tree from a curve is given in Alg. 2.3.

.....

Determine the smallest rectangle with a side parallel to the line segment  $[X_0, X_n]$ , that just covers all the points. This rectangle corresponds to the root of the tree. Pick a point  $X_i$ , that touches one of the sides of the rectangle. This point is termed 'splitting point'. If there are more than one, select the point that is at the maximum distance from the line joining  $X_0$  and the endpoints of the curve. The splitting point  $X_i$  divides the list into two,  $[X_0, X_1, \dots, X_i)$  and  $[X_{i+1}, X_{i+2}, \dots, X_n]$ . Repeat the process for the above two, which will be the left and right sons of the root. The process is repeated till the desired approximation is reached.

.....

### Algorithm 2.3

The half-closed half-open interval facilitates the succeeding computations. A curve and its strip tree are shown in Fig. 2.5.

To determine if two curves intersect, first determine if the corresponding strips do. In case, they do, the procedure is applied recursively till the primitive level is reached.

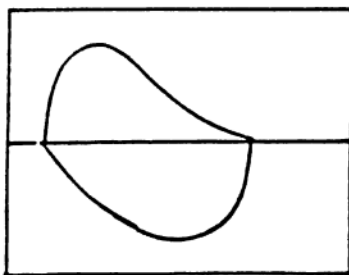


Fig. 2.6 : A closed curve is represented by a pair of strip trees.

```

.....
function STRIP-INT(T1,T2)
/* T1 and T2 are the strip trees representing the two curves */
begin
    case : intersection type of two strips T1 and T2 of
        primitive : return (true)
        null      : return (false)
        possible  : if T2 is the "fatter" strip then
                     return (STRIP-INT(T1,lson(T2)) or
                             STRIP-INT(T1,rson(T2))
                     else
                     return (STRIP-INT(lson(T1),T2)or
                             STRIP-INT(rson(T1),T2))
    endcase.
end.
.....

```

#### Algorithm 2.4

The union of two strip trees may be defined as a strip that encloses both the root strips. If the two curves are defined by  $[X_1, X_2, \dots, X_n)$  and  $[Y_1, Y_2, \dots, Y_m)$ , then these two are concatenated and the strip tree is built for the new list.

A region may be represented by its boundary as shown in Fig 2.6 , by dividing the closed curve into two and storing each of the curves by a strip tree.

##### 2.1.1.4 EDGE QUADTREES

The technique of quadsecting the image space can also be used for storing curvilinear data. But when quadtrees are used for storing curvilinear information, the majority of the quadrants are required to be divided upto the pixel level along the boundaries. The edge quadtree of Shneier [81a] is a step to overcome such high division along the edges. This is an improvement for storing linear feature information for an image (binary or gray-scale) in a manner similar to that used for

storing region information. As in region quadtrees (Section 2.1.2.5), an image containing a linear feature or part thereof is subdivided into four quadrants recursively until quadrants are obtained that contain a single curve that is approximated by a single straight line. Each leaf node contains the following information regarding the edge passing through it:

- magnitude (1 in case of binary image and intensity level in gray-scale)
- direction
- intercept
- directional error term (error induced by a straight line)

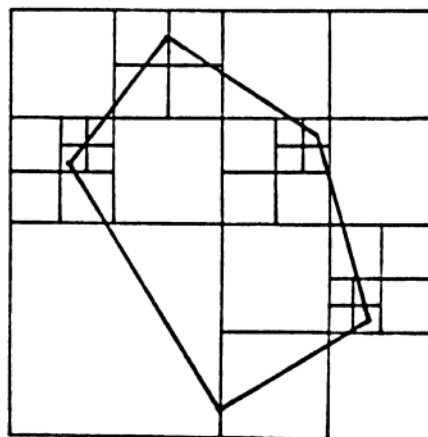


Fig. 2.7 : A polygon and the partition created by the edge quadtree representation.

In case an edge terminates in a node, a special flag is set and intercept denotes the end point of the edge. This method minimizes the storage requirement, although in the vicinity of the vertices, the density of small leaves could be high.

Other data structures which have slight variations to the edge quadtree are the PR quadtree [Orenstein 82], the MX quadtree [Hunter and Steiglitz 79a] and the PM quadtrees [Samet and Webber 85].

Another step in this regard was made by Ayala et al [85]. For the representation of polygons by quadtrees, the node structure of the region quadtree is modified. Besides the usual black, white and gray nodes, they introduced a new type, called edge, i.e., a node type could be one of black, white, gray or edge. Every edge node has a field for storing a pointer to the edge crossing that quadrant. The edges are represented by a triple of real numbers  $(a,b,c)$  corresponding to the three coefficients in the line equation  $ax + by + c = 0$  of the edge. The triplet  $(a,b,c)$  is determined so that the points  $(x,y)$  inside the polygon satisfy  $ax+by+c > 0$ .

Set operations like complementation, union and intersection of two regions, construction of quadtree and display of the polygon from its representation, and conversion from quadtree to edge quadtree are presented. This idea has also been extended to build nonminimal division octrees with node types of black, white, gray, face, edge and vertex. The edge quadtree is shorter than the corresponding quadtree for the same polygon and moreover, the boundary model of the object can be exactly reconstructed.

### 2.1.2 REGION REPRESENTATION

In this subsection, some of the representation schemes are presented that store the interior of an object, as compared to boundary. Owing to the inherent two dimensionality of region information, more schemes based on representing the interior are proposed by various workers.

### 2.1.2.1 SPATIAL OCCUPANCY ARRAYS

The most natural representation for a region on a raster is the membership predicate (or characteristic function of the region)  $p$ , defined by

$$p(x,y) = \begin{cases} 1 & \text{if pixel at } (x,y) \text{ is in the region} \\ 0 & \text{otherwise.} \end{cases}$$

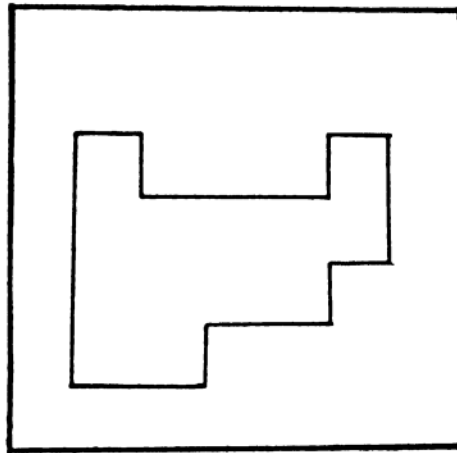
A raster of size  $n \times n$  is represented by a binary square matrix of order  $n$ , where 0 indicates the background and 1, the presence of the object pixel or foreground. When the image is colored or has gray shades, different indices may be given.

Queries regarding point membership problem and area can be answered immediately. Two regions can be merged easily and operations of intersection and union, performed with the use of pixel-wise AND and OR, respectively, are trivial. Space requirement for the storage of an image in an  $n \times n$  matrix is easily seen to be  $O(n^2)$ . But if one employs any programming language that offers bit-wise operators, the space requirement can be reduced drastically. To compute the perimeter, not stored here explicitly, one has to take the aid of contour/boundary tracing algorithms. [Rosenfeld and Kak,82].

### 2.1.2.2 Y-AXIS

To reduce the high space requirement of the spatial occupancy arrays without sacrificing nicety of the algorithms for merging, union and intersection, the Y-axis [Merrill 73] (also called run length code [Rutowitz 68]) representation is proposed. This is a list of lists. Each element in the main list corresponds to a row in the raster image. For each row, as one

moves in the increasing X direction, the X pairs of coordinates at which the image starts and ends are encoded. If there is a lone pixel at  $(x_0, y_0)$ , it is encoded as  $(y_0 \ x_0 \ x_0)$ . A region and its Y-axis representation is shown in Fig. 2.8. Here the first element of each sublist is the Y-coordinate, followed by pair(s) of 'into' and 'outof' X-coordinates.



( (2 2 3) (3 2 5) (4 2 6) (5 2 2 6 6) )

Fig. 2.8 : The Y-axis representation of a region.

The intersection and union are implemented as merge-like operations, which take time linearly proportional to the number of non-empty rows. The union operation amounts to a merge of X-pixels along rows organized within a merge of rows themselves. Two regions A and B and the regions  $A \cup B$  and  $A \cap B$ , with their Y-axis representations are shown in Fig. 2.9.

If the region is thin, long and parallel to the Y-axis, then its representation is not space efficient. In such cases, one may use X-axis representation, a variation of the current scheme, with the obvious meaning. At the cost of convenience, one may work using a combination of X-axis and Y-axis modes, which



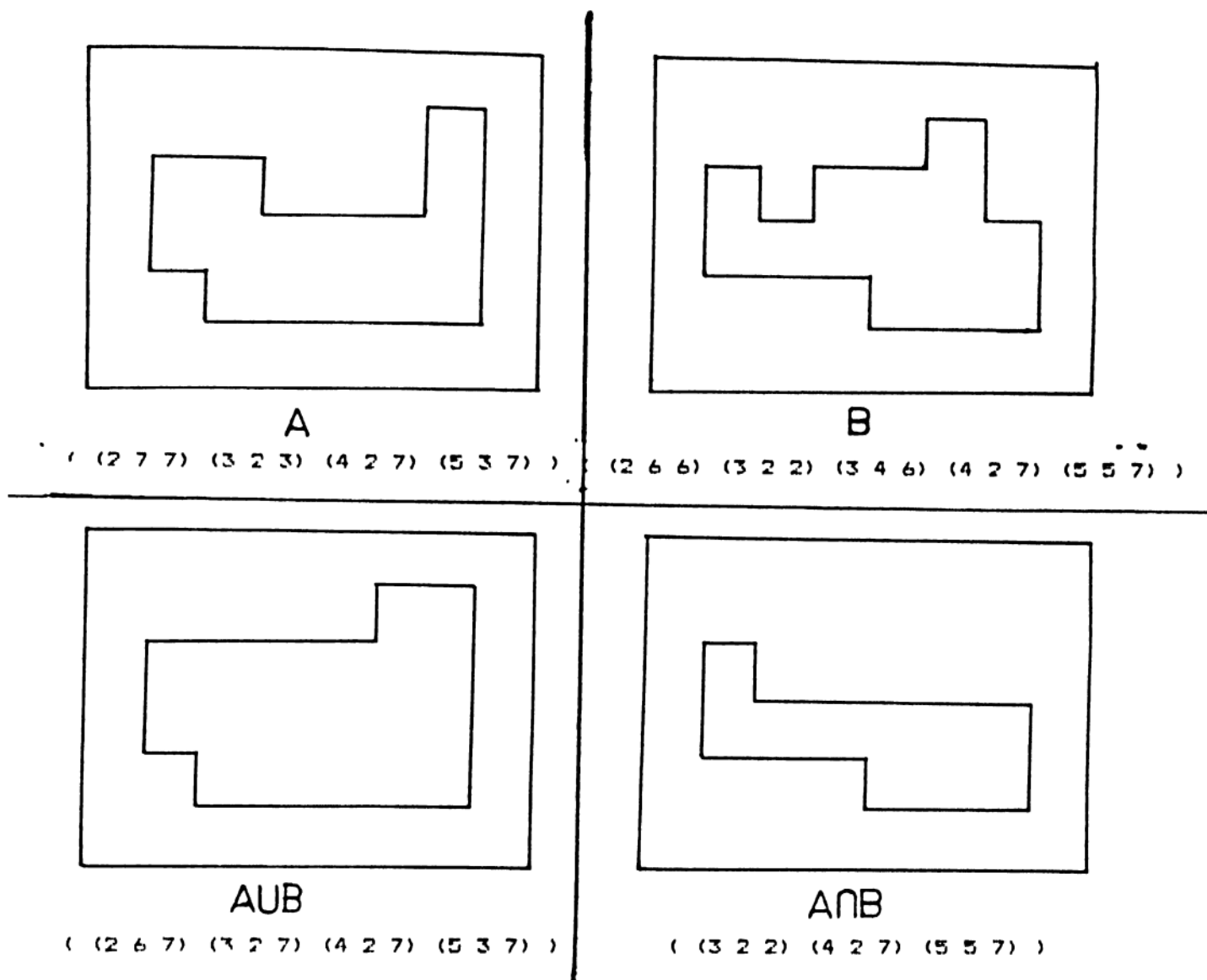


Fig. 2.9 : Two regions A and B and the set operations of  $A \cup B$  and  $A \cap B$  and their Y-axis representations.

presents no conceptual difficulties. Even then, to represent a 'chess-board' like pattern takes much space.

### 2.1.2.3 BINTREES

The bintree [Knowlton 80], is a hierarchical representation of 2D region that uses a binary tree. The region is contained in a spatial occupancy array of size  $N$ , ( $N = 2^n$ , for some  $n$ ).

A bintree is based on recursive subdivision of the image array into halves alternating between X and Y axes. If the array is not consisted entirely of 1's or entirely 0's, it is subdivided into halves, subhalves etc. until the blocks so obtained are of only 1's or 0's. i.e., a block is either completely inside or disjoint from it.

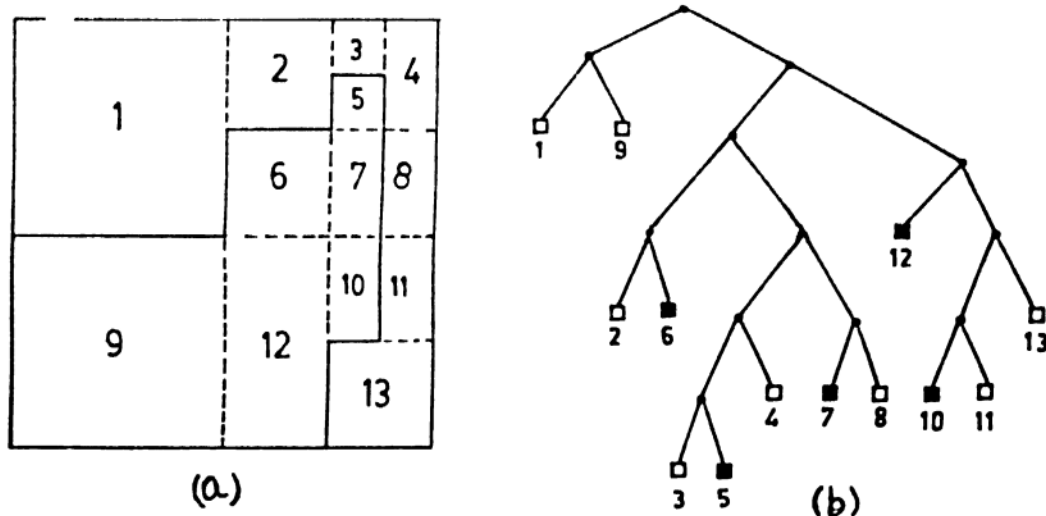


Fig. 2.10 : An object (a) and its bintree representation (b).

The root node corresponds to the entire array. Each son of a node represents half of the region represented by that node. The leaf nodes of the tree correspond to those blocks for which no further subdivision is necessary. A leaf node is said to be black or white, depending on whether its corresponding block is entirely inside or entirely outside the region. A non-leaf node is said to be a gray node.

A node in a bintree has got three fields: two pointers for the sons and a field TYPE, which is one of the three values black, white or gray. To represent an  $N \times N$  image, in the worst case, the height of the bintree would be  $\log N$ , i.e.,  $n$ . A chess-board like image gives rise to a complete bintree.

All the algorithms for answering various queries boil down to tree traversals. The intersection and union of two regions are obtained by traversing the corresponding trees parallelly. Intersection procedure is explained here.

The intersection is done by examining the nodes of each input tree once, visiting nodes in each tree with a stride chosen to keep the two scans 'in step'. Thus if at a certain stage, the next node in each of the trees is a leaf node, the leaves are intersected. If one has a nonterminal node and the other a terminal, then the first scan recurses while the second waits; the intersection in these cases is carried out between nodes of different sizes. If both trees have non-terminal nodes, then both scans recurse.

Complementation is trivial. For each leaf node, the TYPE is changed from black to white and vice-versa. The internal nodes are not modified. Samet and Tamminen[84] present an algorithm for

efficiently labeling the connected components in a region using bintrees.

#### 2.1.2.4 PYRAMIDS

The Pyramid [Tanimoto and Pavlidis 74] is a hierarchical representation of an image organized into layers, each successive layer representing a finer resolution. To represent an  $N \times N$  image array,  $N = 2^n$ , a pyramid is a sequence of arrays  $\{P(i)\}$  such that  $P(i-1)$  is a version of  $P(i)$  at half the resolution of  $P(i)$ , for  $0 \leq i \leq n$ .  $P(0)$  is a single pixel.

In view of the hierarchical structure of the pyramid it is more convenient to think of the pyramid as a complete quadtree [Knuth 68] (Section 2.1.2.5). Starting from the  $N \times N$  image, a recursive decomposition into quadrants is performed till we reach

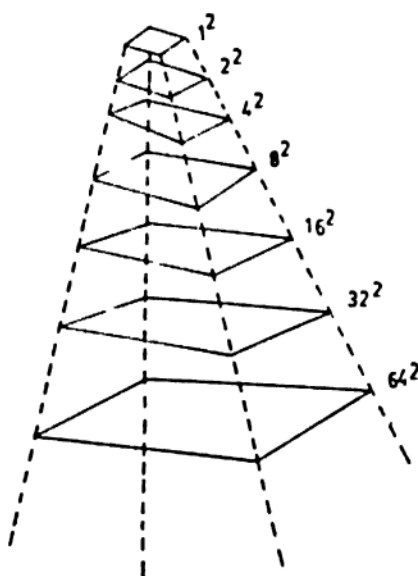


Fig 2.11 : The successive layers of a pyramid. At the top level of the pyramid, the single pixel represents the entire image. At the next level, the four cells represent the partitioning of the image into four equal parts and so on.

the individual pixels. The leaf nodes of the resulting tree represent the pixels. In the non-terminal nodes, the average gray level value of its four sons is stored.

If  $A$  is the area of the image in terms of pixels, then the space requirement for the pyramid is  $4A/3$  and for production of the full set of lower resolution arrays,  $A$  additions and  $A/3$  shifts are necessary. Pyramids are used for feature detection and extraction since they can be used to limit the scope of the search. Once a piece of region of interest is found at a coarse level, the finer resolution levels can be searched. Davis and Roussopoulos [80] used this approach for approximating visual pattern matching. Pyramids are also used for encoding information about curves, lines and edges in an image [Shneier 81a]. Algorithms for edge finding, region growing, and texture analysis have been defined as sequences of parallel operations applied up and down the levels of the tree. The pyramid being a multiresolution representation, enables one to design algorithms that work on several levels of resolution either in parallel or in a controlled sequence.

#### 2.1.2.5 REGION QUADTREES

In the recent times, no other data structure has received as much attention as did region quadtrees. The region quadtree, (hereafter referred to as quadtree) a tree with outdegree four, is based on the principle of recursive decomposition, a divide and conquer method [Aho et al 74]. The term quadtree was first used by Finkel and Bentley [74], where they used the quadtree for partitioning the space into rectangular quadrants.

But the concept of recursively partitioning the space into four equal parts was used earlier by Warnock [69] for implementing a hidden surface elimination algorithm for display of images on the monitor. But later, the meaning of quadtree has changed. The term as it is understood today may be attributed to Klinger [71], Klinger and Dyer [76] who used the term quadtree, whereas it was Hunter [78] who used the specific term quadtree in such contexts as we use today.

The study of quadtrees may be broken up into three parts

- i) pointered quadtrees
- ii) pointerless quadtrees
- iii) semi-pointered quadtrees

#### 2.1.2.5.1 POINTERED QUADTREES

Owing to the inherent recursive process of dividing the region, the tree structure is best suited for storing a quadtree. In this subsection, some of the tree structures with explicit pointers between nodes are considered. Here we mean by quadtree, the pointered quadtree.

The following terminology is being used for the present work.

The term region quadtree, due to Samet [84a], is used to distinguish this type of quadtree from the point quadtree (Section 2.1.0.1). By image, we mean the 2D object we are interested in. The border of the image is the outer boundary of the square, of side  $N$ , corresponding to the array.  $n (= \log_2 N)$  is the resolution. Two pixels are 4-adjacent if they are adjacent to each other either in the vertical or horizontal directions. Two

pixels are 8-adjacent, if the concept of adjacency includes diagonal adjacency too. A black region is a maximal 4-connected set of black pixels. i.e., a set  $R$  such that for any two pixels  $p$  and  $q$  in  $R$ , we can find a sequence of black pixels  $p=p_0, p_1, p_2, \dots, p_l=q$  in  $R$  such that  $p_i$  is 4-adjacent to  $p_{i+1}$ ,  $0 \leq i < l$ . A black region is also called a connected component. A white region is a maximal eight-connected set of white pixels. A pixel is a quadrant of unit length. The boundary of the black region consists of the set of edges of its constituent pixels.

The construction of a quadtree is based on the successive subdivision of the raster image into four equal-sized quadrants. The image is embedded in a square array of size  $N$ ,  $N = 2^n$  for some  $n$ , the resolution, where 1's represent the image and 0's the background. (Here our discussion are limited to binary images i.e., a pixel is either black or white). A block or quadrant is said to be homogeneous if it consists entirely of 1's or entirely of 0's. Otherwise it is said to be non-homogeneous. If the original binary array is non-homogeneous, then it is subdivided into four quadrants. Then each of these four quadrants is checked for homogeneity. The non-homogeneous quadrants, if any, are again subdivided and this process continues till all the quadrants are homogeneous. The root node corresponds to the entire array. Each son of a node represents a quadrant (labeled in order of NW, NE, SW and SE) of the region represented by that node. The leaves represent those quadrants which are homogenous. A leaf node is said to be black or white, according as its corresponding block in the image is made up of 1's or 0's. All internal nodes are said to be gray. Samet [80a] presents an algorithm for

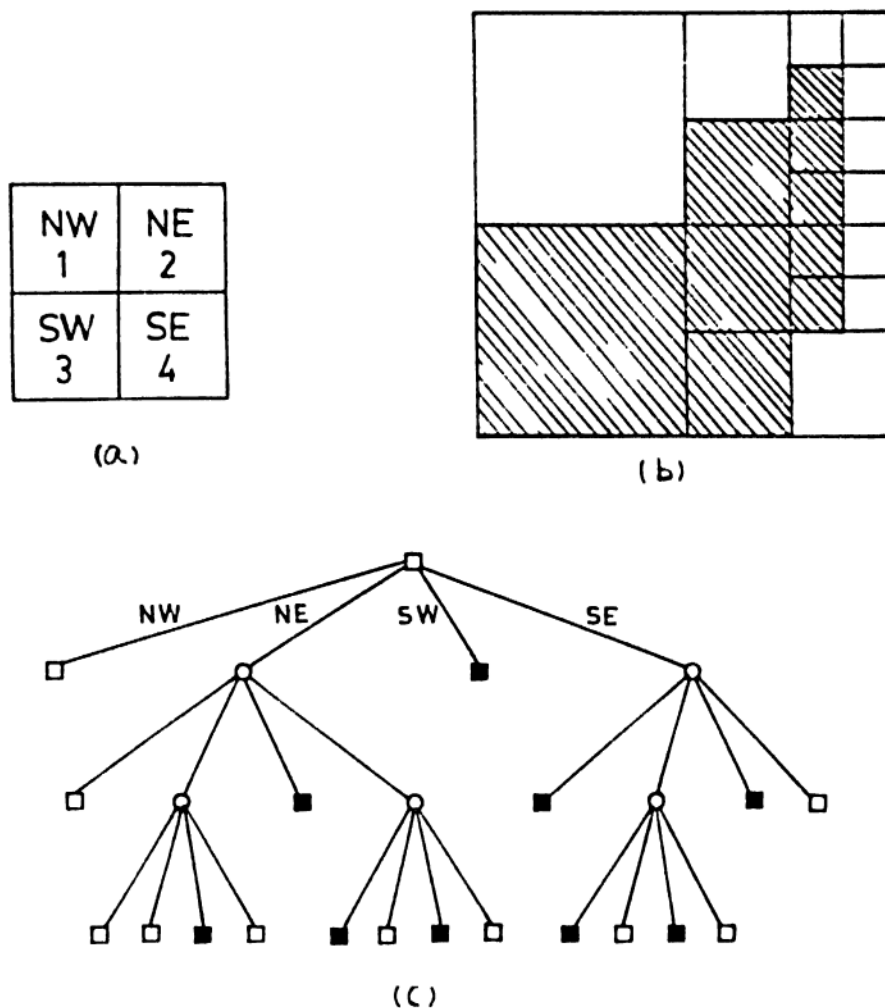


Fig. 2.12 : The ordering of the quadrants (a), an object (b) and its region quadtree (c).

constructing a quadtree from the array representation of a binary image. The execution time is of the order of number of pixels in the image.

A node in a quadtree has the following structure.

| TYPE | FATHER | NW | NE | SW | SE |
|------|--------|----|----|----|----|
|------|--------|----|----|----|----|

TYPE is one of black, white or gray depending on whether the quadrant of the image which the node represents is completely within the image, disjoint from it or neither. FATHER is a pointer to the father node. NW, NE, SW and SE are pointers to the north-west, north-east, south-west and south-east sons of the



node, respectively.

Given a node P and a son I, these fields are referenced as FATHER(P) and SON (P,I) respectively. We use the boolean functions BLACK, WHITE and GRAY to access the TYPE field of a node, which is true if the function name and the value accessed are same.

The quadtree, in view of its tree structure, is well suited for performing many recursive algorithms. This not only makes it easier to program but also allows clear visualization of the concepts. Most of the algorithms, to follow, reduce to tree traversals. Now we present algorithms for set operations (union, intersection and complement), computing areas, moments and centroid, connected component labeling and counting, transformation and determining the perimeter.

### Set Operations

Given two images R1 and R2, represented by the corresponding quadtrees, QT1 and QT2, algorithms are presented [Shneier 81b, Hunter 78, Hunter and Steiglitz 79a] to generate the resulting quadtrees obtained by performing union and intersection of QT1 and QT2. Also, to generate the quadtree of set complement of QT1. It is to be noted here that given complementation and one of intersection and union, the other operations can be derived. For any two sets A and B, it is well known that

$$A \cap B = B - (B - A)$$

$$A \cup B = (A^C \cap B^C)^C$$

$$A - B = A \cap (A \cap B)^C$$

'SONS' is used to denote the set, { NW, NE, SW, SE }, of the four sons of a node.

**Complementation** Generating the complement of an image involves changing black pixels into white and vice-versa. i.e., changing black nodes in the corresponding quadtree to white and vice-versa.

```

-----
function COMPLEMENT (QT1)
begin
  if GRAY(QT1) then
    for I in SONS do
      COMPLEMENT (SON(QT1,I))
    else
      if (BLACK(QT1)) then TYPE (QT1)=white
      else TYPE (QT1)=black.
    end
  end
end
-----

```

#### Algorithm 2.5

If the numerical values of 0, 1 and 2 are given to denote white, black and gray types respectively, in TYPE field, the above algorithm becomes simple, which is presented in Alg. 2.6.

```

-----
COMPLEMENT1 (QT1)
begin
  if TYPE (QT1)=2 then
    from I in SONS do
      COMPLEMENT 1 (SON(QT1,I))
    else
      TYPE(QT1)=1- TYPE(QT1)
    end
  end
end.
-----

```

#### Algorithm 2.6

#### Intersection

This algorithm finds the logical AND of two images R1 and R2 represented by their quadtrees QT1 and QT2 respectively,

traversing the two trees in parallel. When one tree has a black son and the other a non-black son, then the black son is replaced by the corresponding node. If one of tree has a white node, the intersection will have a white leaf at the corresponding position. Finally, if both the trees have gray nodes in the corresponding positions, the algorithm recurses. At the end, the pointer to the root of the resulting quadtree is returned.

```

-----
function INTSECT (QT1,QT2)
begin
  if BLACK (QT1) then return (COPY(QT2))
  else if BLACK (QT2) then return (COPY(QT1))
  INTSECT = CREAT-NODE( ).
  for I in SONS do
    SON (INTSECT,I)=INTSECT (SON(QT2,I))
    FATHER (SON (INTSECT,I))=INTSECT
  end.
  return (INTSECT)
end.
-----

```

#### Algorithm 2.7

The function INTSECT returns a pointer to a node in a quadtree. The function COPY(QT) creates a tree structure identical to QT and returns the pointer of the root of the new tree.

```

-----
function COPY(QT)
begin
  NEWQT = CREAT-NODE ( )
  TYPE (NEWQT) = TYPE (QT)
  for I in SONS do
    if (SON(QT,I))= NULL then SON (NEWQT,I)=NULL.
  else
    begin
      SON (NEWQT,I)=COPY (SON(QT,I))
      FATHER (SON (NEWQT,I))=NEWQT.
    end
  return (NEWQT)
end.
-----

```

#### Algorithm 2.8

## Union

The union algorithm is identical to the intersection algorithm with the roles of white and black nodes swapped. Here too, the trees are parallelly traversed and a decision is made when a leaf node is reached.

```

.....

function UNION (QT1,QT2)
begin
  if BLACK (QT1) then COPY (QT2)
  else if BLACK (QT2) then COPY (QT1)
  UNI = CREAT-NODE ( )
  for I in SONS do
    begin
      SON (UNI,I) = UNION (SON(QT1,I),SON(QT2,I))

      FATHER (SON(UNI,I))=UNI

    end.
  return (UNI)

end.
.....

```

### Algorithm 2.9

The union and intersection algorithms can be generalized to handle any number of quadtrees [Shneier 81b]. The intersection algorithm takes time proportional to the traversal time of the smallest tree whereas the union algorithm requires that of the second largest tree (when more than two trees are considered). The ability to perform set operations quickly is one of the major reasons for the popularity of the quadtrees over other representations [Samet 84a].

Burton et al [87] have presented a single quadtree overlay function that can perform a number of common quadtree operations including union, intersection, difference, masking, copy, complement and map generalization. They assert that a slightly more complicated version of this function ensures the full

advantage of the special characteristics of each of their operations and performs the computations in a manner which is optimal to within a constant factor with respect to both space and time.

### Area and Centroid

Area and centroid [Shneier 81b] for images represented by quadtree is simple to compute. It involves the postorder traversal of a quadtree. For computation of area, as the tree is traversed, the areas of black leaves are accumulated.

```

.....
function AREA (QT,n)
begin
    if GRAY (QT) then
        for I in SONS do
            black_area = black_area + AREA (SON(QT,I),n-1)
        else
            if BLACK (QT) then
                black_area=black_area +  $2^{2n}$ .
            return (black_area)
end.
.....

```

### Algorithm 2.10

The arithmetic mean of the coordinates of all the black pixels gives the centroid, which is same as the arithmetic mean of the centroids of all black leaves. The position of each black block is easy to ascertain from the path that was taken to reach that block.

Both the algorithms have a time complexity that is of  $O(T)$ , where  $T$  is the total number of nodes in the quadtree.

### Geometric Transformations

A major motivation for the development of the quadtree concept is the desire to provide an efficient data structure for computer graphics. Quadtrees are also used in graphics and

animation [Hunter 77, Negroponte 77, Newman et al 84] which are oriented towards construction of images from polygons and superposition of images. Encoded pictures are specially useful for display [Hunter 78] if the encoding is well suited for processing of the image.

Suppose we have the quadtree representation of an image and a general linear transformation

$F : (x, y) \rightarrow (a_1x + a_2y + a_3, b_1x + b_2y + b_3)$  where  $a_i, b_i, 1 \leq i \leq 3$ , are real constants. The aim is to efficiently generate the quadtree of the transformed image. In the transformation  $F$ , setting appropriate values to the constants, the particular transformations of translation, scaling and rotation are obtained.

To achieve this aim, a brute force method would be to decode the quadtree to get the raster image [Samet 84b], apply

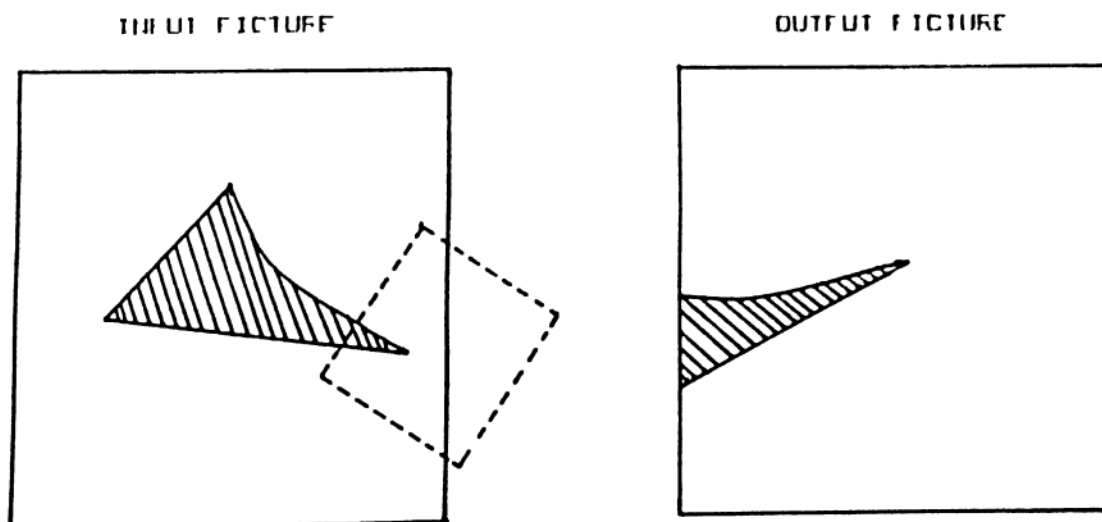


Fig. 2.13 : The inverse of the transformation is applied to the input picture and the part that is of interest is clipped. The dotted square is the inverse image of the transformation.

the transformation pixel-wise and then encode the raster in a quadtree form [Samet 80a, 81b]. This, as easily observed, is a lengthy and time taking process.

Hunter and Steiglitz [79b] have given an algorithm that requires time and space of  $O(T+p)$ , where  $T$  is the total number of nodes in the quadtree and  $p$  is the perimeter of the non-background visible portions of the multicolored image. Here, the image may consist of more than one, possibly overlapping, polygon and may have holes. The algorithm is briefly explained below.

We have an  $N \times N$  square array of pixels, the corresponding quadtree and a general transformation  $F$ . First the nodes which are 'visible' after the transformation are identified by computing the edge of visibility i.e., the inverse image of the output picture boundary. All the visible edges of nodes are stacked. Pop the stack until a leaf-side is found with a visible portion which lies between two different colors or which lies on the edge of the picture and has a foreground-colored leaf. Now, following the boundary of the connected component containing this leaf, generate the polygon formed by edges of leaves. For each polygon, called primary polygon, so outlined, each edge is transformed by  $F$  and the transformed polygon is encoded into a quadtree. Coloring is done by recursive spreading of color from boundary leaves to neighboring leaves. The interior polygons are outlined, transformed and a quadtree for each is generated. Now the interior polygons are cut out of the primary polygon by superposing them as holes in the quadtree for the primary polygon.

The process of popping the stack, finding an inter color edge and building quadtrees, with holes if any, for polygons found is repeated till the stack is empty. Superposing the quadtrees for the transformed polygons will result in the desired transformed quadtree.

In this paper by [Hunter and Steiglitz 79b], the algorithm is presented for a netted quadtree. A net is a linked list whose elements are all the nodes that are adjacent along a given side of a node. But the central idea of the algorithm holds good even for quadtrees, where the neighbors can be determined by neighbor finding techniques, proposed by Samet [82a, 85a].

Peters [85] restricted the class of transformations that transform squares into convex quadrangles ( operations like translation, scaling and rotation own this property), and proposed an algorithm which has  $O(T+m(n+1))$  time and  $O(T+n+m)$  space complexities, where  $T$  and  $m$  are the total number of nodes in the input quadtree, and an intermediate quadtree and  $n$  is the resolution factor. The number of nodes in the resultant quadtree does not exceed  $m$ .

Scaling the image by a power of two is an easy process when using quadtrees, for, it is simply a modification of the resolution [Tanimoto 76]. So is the rotation by multiples of  $90^\circ$ , since it involves a recursive rotation of sons at each level of the quadtree.

Barring the few transformations (scaling by a power of two, translations and rotations in multiples of  $90^\circ$  ), all the other transformations have an inherent shortcoming in them. The resulting transformed picture is only an approximation. Straight



lines are not necessarily transformed into straight lines. This is because of the underlying digitization process. It manifests itself, in any representation scheme involving raster graphics.

### **Perimeter**

The computation of perimeter [Samet 81a] involves the postorder traversal of the quadtree and requires  $O(T)$  time,  $T$ , the total number of nodes in the quadtree.

In the postorder tree traversal, for each black node that is encountered, its four adjacent sides are explored to determine if any adjacent nodes are white. For each of the existing adjacent white nodes, if any, the length of the corresponding shared side is accumulated in the perimeter.

This will result in a certain amount of redundant effort because each adjacency between two black blocks is explored twice, without any further addition to the perimeter. A way out is to explore only for southern and eastern neighbors. In this modified approach [Samet 84a], the northern and western boundaries of the image are never explored, and this is alleviated by embedding the image in white region.

Jackins and Tanimoto [83] have developed an asymptotically faster algorithm for perimeter computation that works for an arbitrary number of dimensions.

### **Connected Component Labeling and Counting**

Connected component labeling is one of the crucial operations for any image processing system. Samet [81c] presents a three step process to label the components.

- (i) Traverse the tree in postorder. For each black node,

say BN, determine all the adjacent black nodes on the southern and eastern sides and assign the same label as that of BN. (It is possible that a node may be assigned the label more than once).

- (ii) Merge all the equivalences generated in the first step.
- (iii) Traverse the tree again and update the labels on the nodes to reflect the equivalences generated in the first two steps.

The algorithm has an average execution time that is of  $O(B \log B)$ ,  $B$ , the number of black nodes in the quadtree. Component counting is a consequence of labeling. The number of different equivalence classes resulting in step (ii) is the required number.

A few remarks are made below on the space and time complexities of quadtrees. To reduce the amount of space necessary to store data through the use of aggregation of homogeneous blocks is the prime motivation for the development of quadtrees. For a simple polygon, i.e., a polygon with non-intersecting edges, of perimeter  $p$  and a resolution  $n$ , the number of nodes in the quadtree is of  $O(p+n)$  [Hunter 78]. Moreover, the quadtree grows linearly in number of nodes as the resolution is doubled, whereas a binary array representation leads to quadrupling of the number of pixels. The amount of space occupied by a quadtree is very sensitive to its orientation and position.

Dyer [82] has shown that the average numbers of white, gray and total nodes in a quadtree representation of  $2^m \times 2^m$  square image in a  $2^n \times 2^n$  region, are each of  $O(2^{m+2+n-m})$ .

If  $B$ ,  $W$  and  $G$  represent the number of black, white and gray nodes respectively, then  $T$ , the total number of nodes is given by

$$T = B + W + G.$$

The following relations hold for any quadtree [Knuth 75].

$$G = (B + W - 1)/3$$

$$B = 3G - W + 1$$

$$W = 3G - B + 1$$

$$T = 4G + 1.$$

Weng and Ahuja [87] (in the context of octrees) have given an elegant proof for  $T$  in terms of  $G$ . We imitate their proof here.

Theorem  $T = 4G + 1$ .

Proof Let  $L$  be the number of leaves.

Suppose that each gray node corresponds to a 4-person game. Originally there are  $L$  players to take part in the tournament. Three players lose after each game. Only the winners participate in the rest of the games. So we have

$$L - 3G = 1$$

$$L = 3G + 1$$

$$L + G = 4G + 1$$

$$T = 4G + 1$$

QED

Hence in a quadtree, there will be an additional  $(B+W-1)/3$  gray nodes besides  $B$  black and  $W$  white nodes, which makes the space requirement of  $O(4/3(B+W))$ .

To reduce the space requirement of the pointer quadtrees, the Autumnal quadtrees is proposed by Fabrini et al [86], in which the leaf nodes in a region quadtree are dropped. Instead,

the leaf node values are stored in place of the pointers to them. The sign bit is used to distinguish pointers from leaf values, with the convention being that positive numbers denote pointers. The Autumnal quadtree gives a space saving of 75% for a complete quadtree.

Since the quadtree of an image is greatly affected by its location, orientation and size, methods have been suggested to overcome this dependency. The effect of translating a region by a unit is shown in Fig. 2.14. The first step in this direction was made by Li et al [82], to eliminate the effect due to translation, to define a normal form of quadtree. Supposing that the size of the image lies between  $2^{l-1}$  and  $2^l$ , the image is moved around in a region of size  $2^{l+1}$  to find a minimal cost quadtree in terms of number of nodes. This can be done in  $O(2^{2l})$  space and  $O(1.2^{2l})$  time. They assert that this quadtree representation is unique for any image over the class of translations.

Chien et al [84] have proposed a scheme, the normalized quadtree representation, which is invariant to rotation, scaling and translation. A normalized quadtree is generated for each object in the image rather than for the entire image. The object is normalized to an object centered coordinate system, with its centroid as the origin and its principal axes as the coordinate axes. Then the object is scaled to a standard size. This ensures that the normalized quadtree of an object is dependent only on the shape of the object but not on its orientation, location or size. This is an information preserving shape descriptor [Pavlidis 78] and hence can be used for object identification.

### 2.1.2.5.2 POINTERLESS QUADTREES

The space requirement of  $4(B+W)/3$  nodes for a pointered quadtree where B and W are the number of black and white nodes respectively, to represent an image in an  $N \times N$ ,  $N = 2^n$ , binary array is high for many real-life applications. Two of the drawbacks of the pointered quadtree are

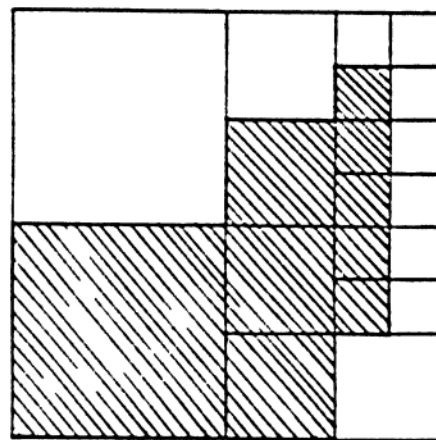
- (i) A large portion of the space requirement is taken up by gray nodes and pointers.
- (ii) Individual leaf nodes are located by following a chain of pointers from the root to the desired node requiring many pointer references.

The pointerless data structure to store a quadtree has been proposed to overcome the above shortfalls.

#### 2.1.2.5.2.1. DF-EXPRESSION

Kawaguchi and Endo [80] represent a tree in the form of preorder tree traversal of the nodes of the quadtree called DF-expression. The result is a word over the alphabet set {B, W, ( )} corresponding to black, white and gray respectively. The DF-expression corresponding to the quadtree is shown in Fig. 2.15.

The original image can be reconstructed from the DF-expressions by using the fact that the degree of any non-terminal node is always four. Kawaguchi et al [83] show how a number of basic image processing operations can be performed using DF-expressions. In particular, centroid computation, rotation, scaling, shifting and set operations, are presented. If the length of the DF-expression is T and if each symbol takes 2 bits for storage, the total requirement will be of  $O(T)$  bits.



(a)

( W ( W ( W W B W B ( B W B W B ( B ( B W B W B W

(b)

Fig. 2.15 : An object (a), and its DF-expression (b).

#### 2.1.2.5.2.2. LINEAR QUADTREES

The reduction of space requirement and elimination of pointers is the prime motivation for the development of linear quadtrees. In this scheme, each quadrant is given a locational numerical code which corresponds to a sequence of directional codes that locate the leaf along a path from the root of the tree. The location of the quadrant manifests itself inherently in the code. The set of only black nodes, stored either in an array or a linked list is termed a linear quadtree. [Gargantini 82a]. Gargantini was the first to propose such a model, but later Abel and Smith [83] and Samet [85b] have proposed similar models and used it for rectangle retrieval, and approximation and compression, respectively.

### Gargantini's Model

There are two phases in the construction of the linear quadtree (LQT).

- (i) all black pixels are encoded and transformed into numerical codes in number system with base 4 i.e., quaternary codes.
- (ii) condense the codes. (as is done in [Samet 81b])

The coordinate system and the quadrant encoding is as shown in Fig.2.16. The encoding process is explained below.

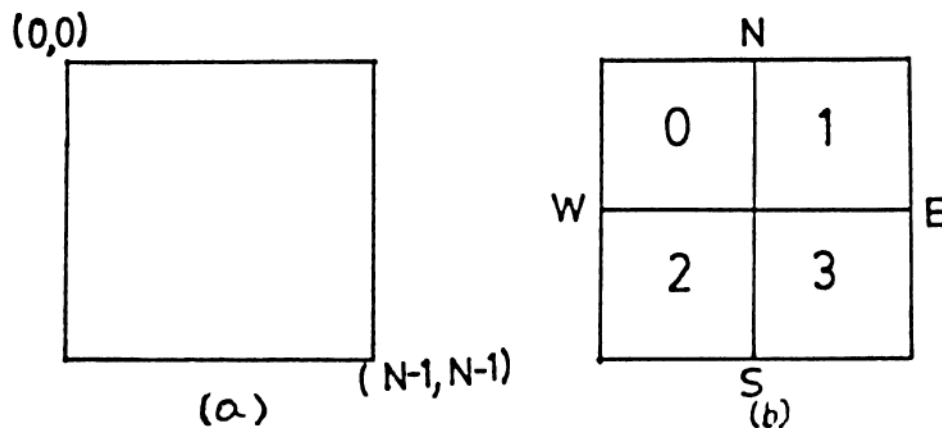


Fig. 2.16 : The coordinate system (a), and the ordering of the quadrants (b).

For a black pixel at position  $(I,J)$ , the binary equivalents of  $I$  and  $J$  are interleaved and then converted to quaternary system. For example, for a black pixel at  $(6,5)$

$$I = 6_{10} = 110_2$$

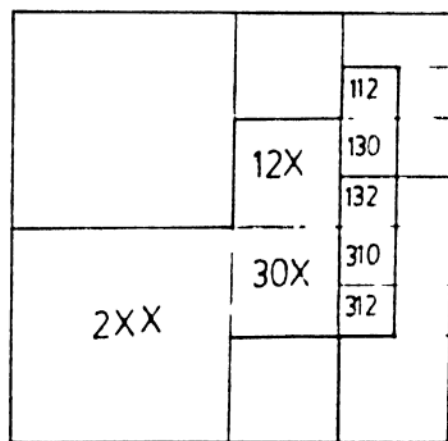
$$J = 5_{10} = 101_2$$

After interleaving the corresponding bits, we have  $111001_2$ , i.e.,  $321_4$ . The location of the pixel in the image space is clear from the quaternary code. The pixel with code 321 is in the SE(3) quadrant at first division, in the SW (2) quadrant in the second

and NE (1) in the final subdivision. A code will have  $n$  quaternary digits,  $n = \log_2 N$ .

After all the black pixels have been transformed into their corresponding quaternary codes, the condensation process starts. If any four pixels or quadrants have the same code but for the last quait ( quaternary digit), the four is replaced by a code of the first  $(n-1)$  quaits followed by 'X'. For instance, if pixels with codes 210, 211, 212 and 213 are all present, they are replaced by 21X. Moreover, if 20X, 21X, 22X and 23X are all present, they are substituted by 2XX and so on. The linear codes sorted in ascending order correspond to the post-order traversal of the black nodes of the corresponding region quadtree.

A region and its LQT based on Gargantini's Model are given in Fig.2.17.



(a)

112, 130, 132, 310,  
312, 12X, 30X, 2XX.

(b)

Fig. 2.17 : An object (a), and its linear quadtree based on Gargantini's model (b).

All the operations on linear quadtrees boil down to manipulation of numbers, which is relatively easy compared to tree traversals using pointers. Given a code, the adjacent quadrant(s), if any, can be found easily [Gargantini 82a].



Let  $T$ ,  $B$ , and  $W$  be as defined before. Let  $NP$  be the total number of black pixels. A linear quadtree can be stored in  $(3(n-1)+2)B$  bits. In terms of number of nodes, the space complexity for a LQT is  $B$  whereas for a regular quadtree it can be as high as  $4nB+1$  [Samet 80b]. Savings of space more than 66% is achieved.

#### Abel and Smith's Model

A similar model for generation of LQT has been proposed by Abel and Smith [83]. The root of the tree, at level 0, has the code  $5^n$  ( $n$  is the resolution). The keys are defined recursively. For a node at level 1, the key  $k$ , can be computed from the key of the father,  $k'$ , by

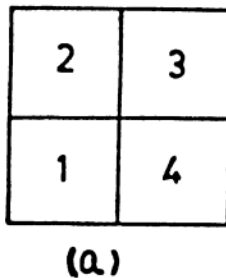
$$k = k' + i5^{n-1}.$$

where  $i$  is 1, 2, 3 or 4 according as the node is the SW, NW, SE or NE son of its father. Three level decomposition with locational keys to base 5 is shown in Fig. 2.18.

The LQT for the image in Fig. 2.17 is

1100, 1320, 1341, 1342, 1410, 1431, 1432, 1441

The locational keys sorted in ascending order gives the preorder traversal of the black nodes of the corresponding region quadtree. Here too, the keys have  $n$  digits. The size and location of a quadrant can be determined from the key. Further, given two keys, it can be easily ascertained if one is contained in the other. Abel and Smith [83] have used this method for rectangle retrieval i.e., to retrieve from a set of spatially defined entities those relevant to a certain subregion. When the LQT is large, for efficient and fast operations, a  $B^+$  tree structure has been proposed [Abel 84].



|   |  |  |  |      |  |           |  |
|---|--|--|--|------|--|-----------|--|
| 1222 1224 1242 1244 1422 1424 1442 1444 |  |  |  |      |  |           |  |
| 1220                                    |  |  |  | 1240 |  | 1420 1440 |  |
| 1221 1223 1241 1243 1421 1423 1441 1443 |  |  |  |      |  |           |  |
| 1200                                    |  |  |  | 1400 |  |           |  |
| 1212 1214 1232 1234 1412 1414 1432 1434 |  |  |  |      |  |           |  |
| 1210                                    |  |  |  | 1230 |  | 1410 1430 |  |
| 1211 1213 1231 1233 1411 1413 1431 1433 |  |  |  |      |  |           |  |
| 1100                                    |  |  |  | 1300 |  |           |  |
| 1122 1124 1142 1144 1322 1324 1342 1344 |  |  |  |      |  |           |  |
| 1120                                    |  |  |  | 1140 |  | 1320 1340 |  |
| 1121 1123 1141 1143 1321 1323 1341 1343 |  |  |  |      |  |           |  |
| 1100                                    |  |  |  | 1300 |  |           |  |
| 1112 1114 1132 1134 1312 1314 1332 1334 |  |  |  |      |  |           |  |
| 1110                                    |  |  |  | 1130 |  | 1310 1330 |  |
| 1111 1113 1131 1133 1311 1313 1331 1333 |  |  |  |      |  |           |  |

(b)

Fig. 2.18 : The ordering of quadrants (a), and the complete numbering of quadrants (b) in Abel and Smith's model, with a resolution of 3.

### Samet's Model

Hanan Samet has used his version of LQT for quadtree approximation and compression [Samet 85b]. The directional codes are obtained by traversing the path from the root to the node. The code for the root is 0 and for the other quadrants, it is defined in a recursive process. The code,  $z$ , can be computed by

$$z = 5z' + i$$

where  $z'$  is code of the father and  $i$  is one of 1, 2, 3 or 4 depending on whether the node is the NW, NE, SW or SE son of its father. The LQT based on Samet's Model for the image in Fig. 2.17 is

3, 13, 21, 63, 71, 73, 111, 113

The nodes sorted in ascending order, gives the breadth-first traversal of the black portion of the corresponding region quadtree.

Samet's model of encoding differs from the others (i.e., Gargantini, and Abel and Smith) in the following ways [Samet 85b].

- (i) For a grid of size  $N \times N$ ,  $N=2^n$ , the locational codes, irrespective of the size of the quadrant, are  $n$  digits long in case of others. In Samet's model, larger the quadrant, smaller is the code.
- (ii) In the 'others' coding scheme, it is more complex to decode the locational code to yield the path from the root of a quadtree to the root of the subquadtree.
- (iii) Increasing the resolution of the image requires recoding in the other models. In particular, their codes must be multiplied by 5 to a power equal to the increase in resolution.
- (iv) Samet's model has the progressive approximation property, i.e., as the list grows, one gets a better approximation of the image i.e., successive nodes in the list lead to a better approximation.

All the operations that can be done using pointer quadrees can be done as well, if not better, by pointerless quadrees.

Algorithms for intersection, union and pairwise difference of two images represented by their respective linear quadrees is given by Bauer [85]. The procedures for translation, rotation and superposition of linear quadrees are presented in [Gargantini 83]. Detection of the connectivity for images represented by LQT

is proposed by Gargantini [82b]. Van Lierop [86] describes how the leaf codes of geometrically transformed pictures can be derived from the original leaf codes, in a time of  $O(MO \cdot (n + \log MI))$ , where  $MI$  and  $MO$  are the number of input and output nodes respectively, and  $n$  is the resolution. But Walsh [88] has presented a nonrecursive translation algorithm, which also has the same time complexity as above, but turns out to execute faster.

Bhaskar et al [88] have presented algorithms for the computation of area, perimeter, center of gravity, moments of images and the basic set operations by means linear quadrees using parallel processing. If there are  $p$  processors and  $n_1$  and  $n_2$  nodes in the two linear quadrees, then the intersection and union can be done in  $O(t(n_1 + n_2)/p + \log p)$ , where  $t$  is the depth of the smallest node. Complementation can be done in  $O(nt/p)$ ,  $n$  is the number of nodes.

Besides the linear quadtree proposed by Samet [85b], Gargantini [82a] and Abel and Smith [83], there are some other schemes which can be said as the close variants of the form. Some of them are presented here.

The Explicit quadtree [Woodwark 82] is proposed to improve the speed of interrogation and modification. Basic operations and an efficient addressing scheme are presented by the author.

Oliver and Wiseman [83a] have defined treecodes, that are based on the depth-first traversal of the quadtree, for performing operations like merging, masking, construction of a quadtree from polygon, rotation, reflection and translation.

The Squarecodes [Oliver et al 83b] is the set of

specifications of the colour, size and position of image square. In the conventional quadtree model, the image is broken up into squares, that have sides which are powers of 2 long, whereas in the squarecode representation, the image is broken up into arbitrary square areas, each one of uniform colour. Algorithm for operations of translation, scaling,  $90^\circ$  rotations, restricted reflection and rotation through an arbitrary angle about the image center are presented.

For progressive transmission of images using linear quadtree, largely used in satellite image processing, a method of storing linear quadtrees, termed 'P-compressed quadtrees' was proposed by Anedda and Felician [88]. This is based on the Gargantini's model. In the new scheme, all the distinct prefixes are stored once; each of them will be followed by the number of pixels having that prefix and by the corresponding suffixes. For example, the five locational codes 001312010, 001312011, 001312020, 001312023 and 001312031 are replaced by 0013120, 5 and 10, 11, 20, 23, 31.

Unnikrishnan et al [87] have presented an improvement over Gargantini's model of linear quadtrees called Linear Hierarchical Quadtree, LHQT. Here, 4 is used for the 'don't-care' digit. For a grid of size  $N \times N$ ,  $N=2^n$ , there will be an extra  $n$  arrays, where the  $i^{\text{th}}$  array will contain the locational codes of quadrants of size  $2^i \times 2^i$ . Since the level of hierarchy indicates the size of the black nodes, the digit 4 is redundant and hence deleted. As a consequence, the codes at level  $k$  will contain  $(n-k)$  digits only. These modified codes are termed linear hierarchical Q-

codes. An algorithm for labeling the connected components by bottom-up approach is presented.

The Threaded linear hierarchical quadtrees, TLHQT [Unnikrishnan et al 88] is an enhancement of LHQT. In the TLHQT, links to neighbor nodes along the four directions at the same or higher levels of hierarchy are provided. Connected component labeling, perimeter and Euler number computation are addressed in this paper.

#### 2.1.2.5.3 SEMI POINTERED QUADTREES

The fully pointered quadtrees offer the maximum flexibility, for, these quadtrees can be traversed in any order, but they require the maximum storage space. On the other hand, the pointerless quadtrees are the most compact, but have to be traversed in the order of their creation, which reduces the speed of some algorithms. In between these two extremes of the space-time trade-off enter the semi-pointered quadtree representations.

##### 2.1.2.5.3.1 ONE-TO-FOUR QUADTREES

The one-to-four quadtree, or just one-to-four, proposed by Stewart [86] reduces the number of pointer fields in a complete quadtree. In the one-to-four, a node has five fields : four fields contain the node types of the four sons, (as against pointers to the four sons, as in pointered quadtrees) and the fifth field is a single pointer to all the sons within that quadrant. See. Fig 2.19. The one-to-four is so called because one pointer is used to determine where the sons' records are for the four quadrant descriptors. The node structure is shown below.

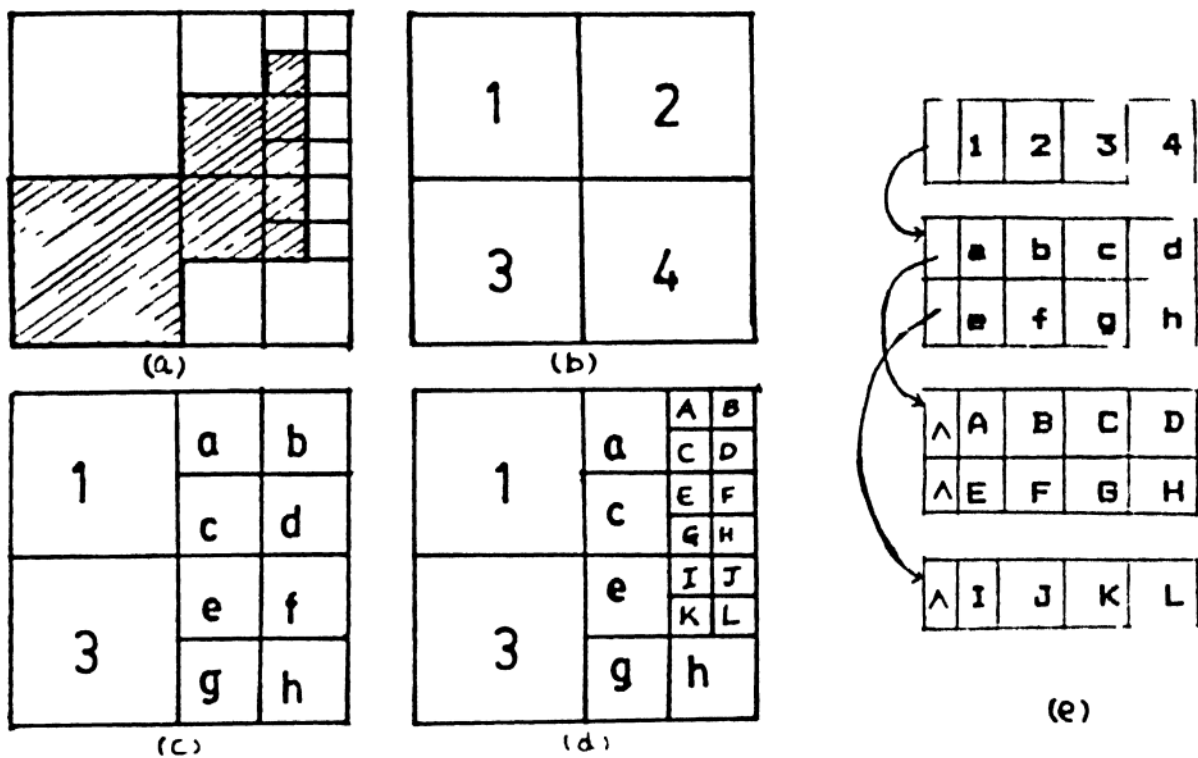


Fig. 2.19 : A 2D object (a), the first division (b), second division (c) and the final division (d). In (d) all the quadrants are homogeneous. The one-to-four quadtree is shown in (e).

Node types of the

|                                 |           |           |           |           |
|---------------------------------|-----------|-----------|-----------|-----------|
| pointer to sons<br>of quadrants | NW<br>son | NE<br>son | SW<br>son | SE<br>son |
|---------------------------------|-----------|-----------|-----------|-----------|

Address of the record for the son  $i$  can be computed using the expression

$$\text{pointer} + [\text{record length} * i]$$

where  $i$  is 1, 2, 3 or 4 according as the son is NW, NE, SW or SE son. Here one pointer replaces 16 pointers and hence in a complete quadtree gives a saving of 93% memory.

This structure is quite useful when the quadtree is complete, but this fails when the quadtree is not. In such cases, the following remedies are helpful.

- (i) Inspecting all the preceding quadrant descriptors to determine the number of sons' records that are there in front of the desired record, at the cost of extra time for traversal.
- (ii) By padding out the one-to-four with extra empty records, the sons' records' address calculation will remain simple. This padding results in each pointer pointing to four records whether they are used or not. Space is wasted, but time is gained for various operations.
- (iii) With a premium of very little space and no additional time, a way out is to add one more field, 'offset', which gives the offset from the pointer for each son's record. And the address of a son's record is given by
 
$$\text{pointer} + [\text{quadrant offset} * \text{record length}].$$



The modified node structure is

Node types of the

|        |                                |           |           |           |           |
|--------|--------------------------------|-----------|-----------|-----------|-----------|
| offset | pointer to sons<br>of quadrant | NW<br>son | NE<br>son | SW<br>son | SE<br>son |
|--------|--------------------------------|-----------|-----------|-----------|-----------|

This tree structure maintains traversal flexibility and reduces memory requirement. Algorithms for one-to-four to raster conversion,  $90^\circ$  rotations and reflections about the two diagonals and the horizontal and vertical axes through the center point are presented.

#### 2.1.2.5.3.2 GOBLIN QUADTREE

The goblin quadtree [Williams 88] is a new and simple data structure for storing spatial information, consisting of a root and a number of branches. Each branch is stored as a record in a direct access disc file. A branch contains five fields : the node values for the NW, NE, SW and SE sons and a pointer. For the node values, a positive value indicates a terminal node, with the value being either an attribute or a pointer to a list of attributes. A negative value indicates a non-terminal node, with the magnitude being the dominant value of the quadrant it represents. See Fig. 2.20.

The pointer field contains the number of branch that would be processed next, if the detail for the current branch were to be skipped. If all the four quadrants of a given branch are homogeneous, the pointer will point to the next branch. If skipping detail for a particular branch would complete the traversal, the value of the pointer is irrelevant. Such a pointer is given a value of one greater than the final branch number.



values. The advantages of goblin quadtree over autumnal quadtrees are

- (i) the average values are stored one level higher in the goblin quadtree, so a truncated traversal requires a reading of far fewer records.
- (ii) goblin quadtree does not mix pointers and leaf values, so that space is not wasted when the pointer requires more bits than the leaf value.

Oliver et al [84] have proposed an algorithm that converts a linear quadtree into run length coding via an intermediate semipointered quadtree called sextree. In the sextree, the pointers allow for quadrants 0 and 1 or 2 and 3 to be missed out while walking over the tree, which is not possible with a linear quadtree. Since any scan line either passes through quadrants 0 and 1, or 2 and 3, not both or any other combination, this is feasible.

#### 2.1.2.6 SKELETONS AND MEDIAL AXIS TRANSFORM

Most of the representation schemes are highly dependent on the location of the image in the region i.e., a slight translation of the image may lead to a drastic modification of the representation. A structure that is invariant to the location of the image is the skeleton [Pfaltz et al 67]. Here the image is regarded as a union of maximal neighborhoods of its points, and can be specified by the centers and radii of these neighborhoods. The set of centers is termed skeleton, see Fig. 2.21, since the locus of centers of maximal neighborhoods often takes the form of centrally located stick figure. It may be observed that the

neighborhoods may overlap.

Depending on the nature of the figure, one of the following metrics may be used to specify the neighborhoods.

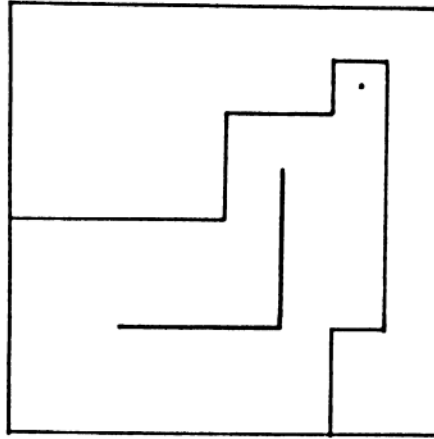


Fig. 2.21 : The skeleton of a 2D object.

### Euclidean Metric

Let  $p$  be  $(p_x, p_y)$  and  $q$  be  $(q_x, q_y)$ . Then

$$d_E(p, q) = [(p_x - q_x)^2 + (p_y - q_y)^2]$$

The ball,  $d_E(p, x) \leq r$ , is a circular disc centered at  $p$  and radius  $r$ .

### Absolute Value Metric

$$d_A(p, q) = |p_x - q_x| + |p_y - q_y|$$

The ball,  $d_A(p, x) \leq r$ , is a diamond centered at  $p$  and side length  $r\sqrt{2}$ . This is also called city block distance.

### Maximum value metric

This is also called chess board metric.

$$d_M(p, q) = \max \{ |p_x - q_x|, |p_y - q_y| \}$$

The ball,  $d_M(p, x) \leq r$ , is a square centered at  $p$  and side  $2r$ .

The set of points comprising the skeletons and their associated radii is called the Medial Axes Transform, MAT. The

operations such as translation and scaling are easy. The former involves modification of the skeletons and latter the radii. The Algorithms for answering the point-membership problem and for performing intersections and union of two images represented by their MATs is given by Pfaltz and Rosenfeld [67]. Wu et al [88] present algorithms for the computation of geometric properties using MATs by parallel processing.

#### 2.1.2.7 QMAT

The skeleton and MAT concepts used in traditional image processing representations, when adapted to the quadtree representations, yield the new data structure, Quadtree Medial Axis Transform or QMAT [Samet 83]. In a QMAT, black nodes in the original quadtree are allowed to expand to absorb adjacent smaller black nodes. At times, they overlap the boundary of the grid. An object and its QMAT are shown in Fig. 2.22.

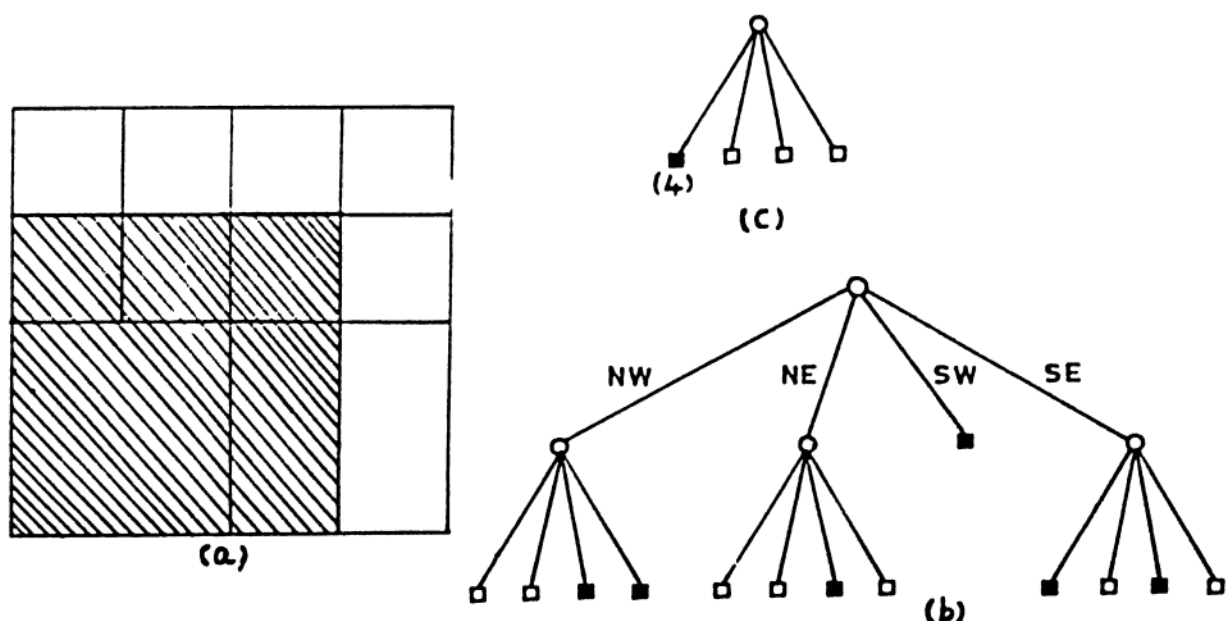


Fig. 2.22 : An object (a), its quadtree (b) and QMAT (c). In QMAT representation, the radius is given in parentheses below the node.

Similar to the skeleton of MAT, there is a quadtree skeleton in QMAT. Before defining the quadtree skeleton some definitions are in order.

Since the ball centered at  $p$  and radius  $r$ , i.e.,  $d_M(p, x) \leq r$ , is a square in chess-board metric, it is well suited over other metrics, for the quadtree operations [Samet 82b]. The function DIST, yields for each black block in the quadtree the chess-board distance transform from the center of the block to the nearest pixel which is on a black-white border i.e., if  $p$  is the center of the black block  $B$  and  $z$  is a point on the border,  $\text{bord}(W)$ , of the white block  $W$ , we have

$$F(B, W) = \min_{z \in \text{bord}(W)} d_M(p, z)$$

i.e.,  $F(B, W)$  gives the distance between  $p$ , the center of  $B$  and the nearest point on border of  $W$ . The minimum taken over all such  $W$ 's gives  $\text{DIST}(B)$  i.e.,

$$\begin{aligned} \text{DIST}(B) &= \min_W F(B, W) \\ &= \min_W \min_{z \in \text{bord}(W)} d_M(p, z) \end{aligned}$$

$\text{DIST}()$  for a white block is defined to be zero. Now we define quadtree skeleton.

Let  $BB$  be the set of all black blocks in the image. For each  $B_i \in BB$ , let  $S(B_i)$  be the part of the image spanned by a square with side width  $2 \cdot \text{DIST}(B_i)$ , centered about  $B_i$ . The set  $T$ , of black blocks, having the following properties is the quadtree skeleton.

- (i)  $\text{area}(BB) = \bigcup_{t_i \in T} S(t_i)$
- (ii) for any  $t_i \in T, \nexists B_j \in BB, (b_j \neq t_i), \ni S(t_i) \subseteq S(B_j)$ ,
- (iii)  $\forall B_i \in BB, \exists t_i \in T \ni S(B_i) \subseteq S(t_i)$

Samet [82b] has shown that the quadtree skeleton of an image is unique. The QMAT of an image is the quadtree whose black nodes correspond to the black blocks comprising the quadtree skeleton and their associated chess-board distance transform values. All the remaining nodes in QMAT are white and gray with distance value zero.

A QMAT results in a partitioning of the image into set of non-disjoint squares having sides whose lengths are sums of powers of two, whereas, in the region quadtrees the image is made up of a set of disjoint squares having sides whose lengths are powers of two. The number of nodes in a QMAT for a region is less than that of the corresponding quadtree. For a certain class of images, the QMAT requires a minimum number of nodes regardless of the image resolution. Specially in cases where  $2^n \times 2^n$  grid consists of square with side  $2^{n-1}$ . For the QMAT representation, the number of nodes necessary to represent an image is not as shift sensitive as in the region quadtree.

#### 2.1.2.8 TID

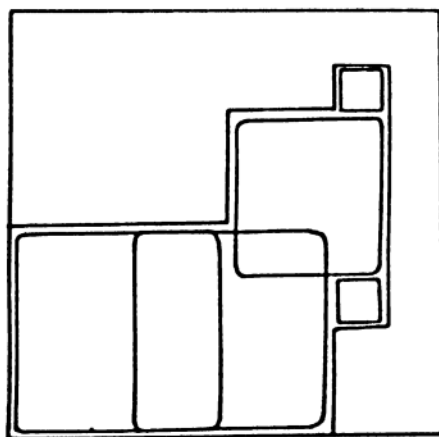
A data structure that is not sensitive to the location of the image, is the Translation Invariant Data structure, TID [Scott et al 86]. This is a slight variation of the MAT. In the MAT, maximal blocks are stored by their centers and radii. A TID is made up of maximal squares and each square is stored as a triplet  $(i,j,s)$ , where  $(i,j)$  is the north west corner of the square and  $r$  the length of the side. In Fig. 2.23, an object and its TID are presented.

The following steps are involved in the construction of a

TID of an image.

- (i) For each black pixel, compute the distance of the nearest white pixel. Let  $D(i,j)$  be the distance from  $(i,j)$  to the nearest white pixel.
- (ii) Locate all maximal black squares. A maximal black square is defined to be a square not contained in any other black square. The largest square centered  $(i,j)$  is of the odd order of size  $(2*D(i,j)-1)$ .
- (iii) Check for redundant maximal black squares and eliminate if any.
- (iv) Identify each non-redundant maximal square by the triple  $N(i,j,s)$ , where  $(i,j)$  is the north-west corner of the square and  $s$  is the size.

In the TID, as with MAT, answering the point membership question is easily done. Translation and scaling are also easy. Rotation by  $90^\circ$  clockwise involves changing  $(i,j,s)$  to  $(-j,i+s,s)$  for each maximal block. Union of two regions represented by their TIDs is also direct.



(a)

$(2,7,1)$ ,  $(3,5,3)$ ,  $(5,1,4)$ ,  
 $(5,3,4)$ ,  $(6,7,1)$ .

(b)

Fig. 2.23 : A 2D object (a), and its TID (b).



Scott and Iyengar [86] have shown that on the average, TID provides a 56% space reduction over linear quadtrees and an 86% over quadtrees, forests of quadtrees and QMATs. They prove that the number of nodes required to store a TID does not exceed the minimum number required by optimally located quadtrees, forest of quadtrees or QMATs.

Scott and Iyengar [84] note that the time required to construct a TID is of  $O(rc \log (\min(r,c)))$ , where  $r$  and  $c$  are the number of rows and columns in the embedded region. In principle, the TID for a  $2^n \times 2^n$  image requires  $3n$  bits for each maximal square, but this space requirement can be reduced by having the minimal number of maximal squares.

Kim et al [83] designed a method, rectangular coding, that partitions the image into disjoint rectangles. Each rectangle is stored by means of the coordinates of the corner nearest to the origin of the coordinate system, its length and width.

## 2.2 3D OBJECT REPRESENTATION

The 3D object representation is crucial for computer vision. A great majority of objects in the real life are 3D and to make the robot an efficient companion to humans, 3D objects or the important attributes of it should be represented so that they may be learned, compared, retrieved and used for image analysis and pattern recognition. After the advent of 2D graphics, the next natural step is to explore the storage, the manipulation and the display of 3D objects in a realistic manner. In fact, this was necessitated by the introduction of present data collection techniques such as Computer Axial Tomography (CAT), Scanning Electron Microscope (SEM) and Digital Stereoscopy. These methods make possible the computation of 3D structure of scenes, ranging from inner parts of the human body to the rock microstructures in the form of a 3D array of numbers. Developing the algorithms for collection, storage, display and manipulation of these images has revealed the need for developing new data structures, and more generally, for developing spatial-knowledge representation schemata.

The task of representing the 3D objects is more intricate than that of the 2D objects, mainly because of the extra dimension involved. On the face of it, using spatial occupancy arrays, it requires  $O(N^2)$  space in case of 2D, whereas, it is

of  $O(N^3)$  for 3D. Besides the phenomenal rise in the space requirement, we look at a 2D object, from outside the 2D space containing it, which enables us to grasp the complete information. But the same cannot be done to 3D objects.

In this section some of the 3D object representation schemes are presented. Requicha [80] has classified the 3D object representation schemes into six categories: primitive instancing, spatial enumeration, cell decomposition, constructive solid geometry, sweep representation and boundary representation. In primitive instancing, components of objects are defined parametrically. A shape type and a limited set of parameter values specify an object. This is best suited for well-defined objects i.e., objects that obey a fixed mathematical equation. The details of the other categories will be considered.

Specific merits and demerits of each have been studied and presented by Requicha et al [80]. In the remaining part of this section, an object is assumed to be a three dimensional one. The object is made up of volume elements or voxels, (Meagher[82a] calls them 'obels' for object elements). Here objects are assumed to be monochrome and voxels inside the object have a value of 1 and outside, 0.

### 2.2.1 SPATIAL ENUMERATION

In this scheme, an object is represented by a list of cubical cells which it occupies.

#### 2.2.1.1 SPATIAL OCCUPANCY ARRAYS

Just as in the 2D case, this is the most simplest method to represent a 3D object. In low resolution tasks where the

objects are highly irregular, this is quite useful.

Using this method, it is easy to perform the geometric operations, set operations, and the computation of volume and moments. Further, by the use of this method, it is easy to generate 'slices' i.e., cross-sections of objects. The spatial occupancy arrays are used in CAT because, here, the objects are highly irregular. And the fast generation of slices facilitates on-line diagnosis. With the declining cost of computer memory, one can afford this representation model, in spite of its high space requirement of  $O(N^3)$ .

#### 2.2.1.2 MAT

The Medial Axis Transform (MAT) representation, besides being compact, is translation invariant. In the 3D version of the MAT, the object is decomposed into a collection of overlapping balls [O'Rourke et al 79]. The metric used is the  $d_M$ , the maximum-value metric, where balls are cubes. Given the centers and radii, it is possible to generate the boundary representation by choosing those voxels that are not in the interior.

Since the 'medial axis' in the case of a 3D object is not a space curve, it may be referred to as the medial layer [Srihari 81]. A voxel  $v$  is in the medial layer if

$$\text{card} \{ u | (u,v) = \min_{w \in B(S)} d_M(v,w) \} > 1$$

where  $S$  is the object and  $B(S)$ , its boundary. This definition means that  $M(S)$ , the medial layer, consists of those voxels of  $S$  whose distances from  $S^C$ , the complement of  $S$ , are local maxima.

Thus,  $S$  is the union of all maximal digital balls centered at the voxels of  $M(S)$ .

### 2.2.1.3 OCTREES

The cube of side  $N$ ,  $N=2^n$ , enclosing the object is our universe. The recursive partitioning of this universe into eight equal octants, suboctants and so on, till the ultimate octants so obtained are homogeneous is the underlying concept for the octree. This is the extension of the quadtree to three dimensions and the terminology of quadtrees is carried over to the octrees. The octree was first briefly described by Hunter [78], and later was independently developed by Reddy et al [78], Jackins et al [80], Meagher [80] and Srihari [80]. The octree is also referred to as octtree, oct-tree [Jackins et al 80] and octal-tree [Srihari 80].

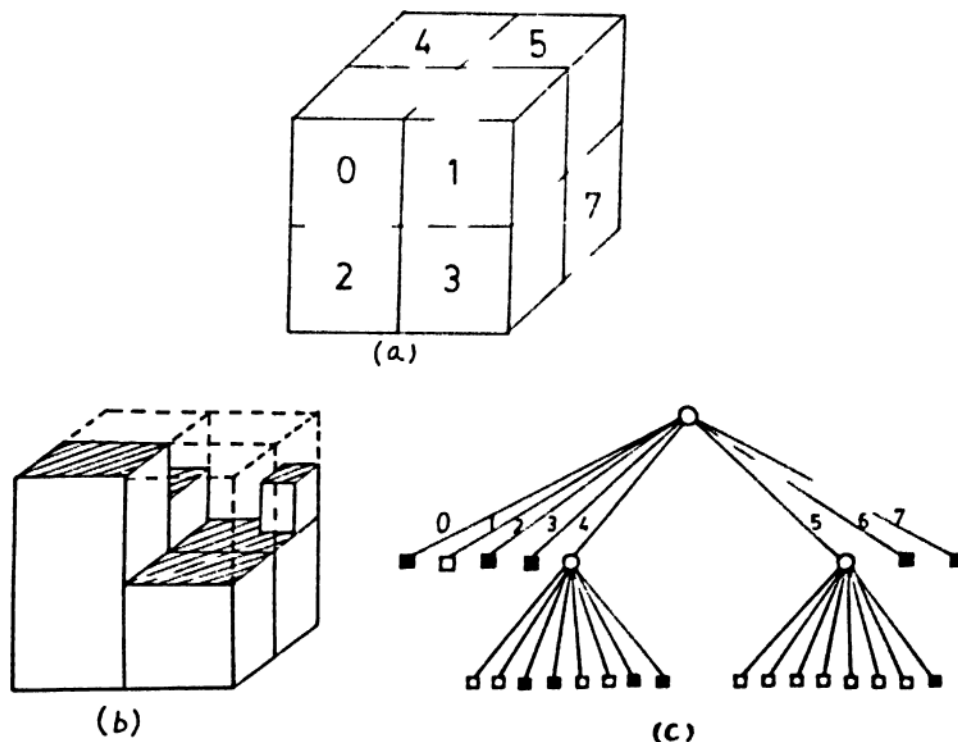


Fig. 2.24 : The ordering of octants (a), a 3D object (b) and its octree (c).

The octree is formally defined now. In the universe, as defined above, each voxel has a value 0 or 1 depending on whether it is outside or inside the object. A cube is homogeneous if all the voxels that are inside have the same attribute. If the universe is non-homogeneous, it is divided into eight equal octants. All the non-homogeneous octants are divided again. This process is repeated till all the octants are homogeneous or the voxel level is reached. In the tree structure, the root corresponds to the universe, its eight sons, to the eight octants obtained at the first division and so on. The height of the octree, in the worst case, is  $n$ . Let  $T$ ,  $B$ ,  $W$  and  $G$  denote the number of the total, the black, the white and the gray nodes in the octree, respectively. Then the following relations hold [Knuth 68].

$$T = 8G + 1$$

$$B = 7G - W + 1$$

$$W = 7G - B + 1$$

$$G = (B + W - 1)/7.$$

Further for a cube of side  $m$ ,  $m > 3$ , at any position and with any orientation, the following relations hold [Weng et al 87].

$$B \leq 24.25.4^m - 200.2^m + 1454.$$

$$G \leq 5.76.4^m + 17.2^m + 8n + 76m - 277.$$

$$\text{and } T \leq 47.4^m + 136.2^m + 64n + 608m + 2215.$$

An object and the corresponding octree are given in Fig 2.24. The octants are ordered as shown in Fig. 2.24a.

Meagher[80] asserts that the memory and processing computation required for the construction of an octree are on the order of the surface area of the object. Many operations on

objects benefit from the tree structure of an octree and can be simply implemented as tree traversal algorithms. The hierarchy of the data structure, spatial accessibility and pre-sorted nature of the octree simplify operations like detecting intersection among objects, hidden surface elimination, locating a point or block in space, connected component labeling and neighbor finding. The octrees find their application in solid modeling, computer graphics, CAD, computer vision, image processing and robotics.

The set operations of union and intersection of two objects and set complement, can be carried out as described for quadrees in Section. 2.1.2.5, with minor amendments. But the quintessential concept of traversing the two trees in parallel remains the same. Meagher [82b] has developed efficient algorithms without the use of floating-point operations, multiplications and divisions. The computation of moments, volume, surface area, component counting, connectivity labeling and neighbor finding can easily be extended from the quadtree and hence not described here.

Given a transformation and the octree of an object, the aim is to build the octree of the transformed object. Separate algorithms exist for translations [Meagher 82a, Jackins et al 80] and rotations [Meagher 82a, Weng et al 87]. Weng et al [87] have given a general algorithm that takes care of both translation and rotation. Their algorithm is described below.

Let  $X$  and  $X'$  be the coordinate vectors before and after the transformation,  $T$ , the translation vector and  $R$  be a three

dimensional rotation matrix for rotation by an angle about the unit vector  $n = (n_x, n_y, n_z)$  through the origin.

Then

$$X' = RX + T$$

Where

$$R = \begin{pmatrix} (n_x^2 - 1)a + 1 & n_y n_x a - n_z b & n_x n_z a - n_y b \\ n_y n_x a - n_z b & (n_y^2 - 1)a + 1 & n_y n_z a - n_x b \\ n_z n_x a - n_y b & n_z n_y a - n_x b & (n_z^2 - 1)a + 1 \end{pmatrix}$$

$$a = 1 - \cos\theta$$

$$b = \sin\theta$$

Let  $OT_1$  and  $OT_2$  represent the source octree and target octree respectively. In  $OT_2$ , a voxel is treated black if and only if its center is on or inside some displaced black cube from  $OT_1$ . The accuracy is lost by coloring partially occupied voxels as black or white. If they are colored white, they may create holes in the transformed object. On the other hand coloring them black would tend to increase the volume.

Informally, the algorithm is described below. For each black leaf, BL, in  $OT_1$ , the following are performed to generate  $OT_2$ .

- (i) generate the new position and orientation of BL. If this is outside of the universe, report error and stop.
- (ii) generate the nodes in  $OT_2$  by testing their corresponding upright cubes for intersection with the transformed BL.

An octant/cube is upright if its sides are parallel to the coordinate axes. Else, it is tilted. The intersection test leads to three cases, each leading to different action about the nature of the target tree node(s).



- (i) no intersection: the upright cube is unoccupied and is left alone
- (ii) the upright cube is inside the transformed BL: the upright cube is occupied and hence colored black.
- (iii) otherwise: generate all the eight children of the upright cube and carry out the intersection test recursively. If all the eight children are of a single color, delete the children and make the father the same colour.

A formal description of the algorithm follows.

```

-----
function FDLEAF (node)
/* traverses OT1 in post order and for each black node checks if this leaf
node goes out of the universe after displacement */

begin
  if BLACK (node) then
    if (out-of-space) then error-stop
    else INTERSECT (node).
  else
    if GRAY (node) then
      for each child, i of node do FDLEAF (i).
end
-----

```

#### Algorithm 2.11

In a universe with voxel resolution  $n$  the level of the root node is defined to be  $n$  and for a non-root node, its level is one less than the level of its father.

For the functions **INTERSECT** and **PARTIAL** to be time effective, a bounding sphere circumscribing the transformed **OT<sub>1</sub>\_node** is used to help identify the target octants that are inside or intersect the source octant.

```

-----
function INTERSECT (OT1_node,OT2_node)
/* l1 and l2 are levels of OT1_node and OT2_node respectively */
begin
  if l1 < l2 then
    if PARTIAL (OT1_node,OT2_node) then
      for each child i of OT2_node do INTERSECT (OT1_node,i)
    else
      if l2 = 0 then
        if (center of OT2_node is inside OT1_node) then
          TYPE (OT2_node) = BLACK
        else TYPE (OT2_node)=WHITE
      else
        if INSIDE (OT2_node,OT1_node) then
          TYPE (OT2_node) =BLACK
        else if PARTIAL (OT2_node,OT1_node) then
          for each child i of OT2_node do INTERSECT(OT1_node,i)
          CONDENSE (OT2_node)
end.
-----

```

#### Algorithm 2.12

The average space and time of the algorithm are both bounded by the number of nodes in the source octree multiplied by the resolution of the universe.

Using octrees, it is easy to display objects by eliminating the hidden surfaces. Since the octree maintains all the elements of an object in a spatially pre-sorted order, display of objects after hidden surface removal is rendered easy. One can display the object by following a particular visiting sequence of octants. There are two types of visiting sequences front-to-back [Meagher 82b, Gibson 83] and back-to-front [Doctor et al 81 and Frieder et al 85]. In the front-to-back approach, octree nodes closer to the viewer are projected prior to those that are farther away. If a region of the image plane is painted, any other nodes projected onto it are ignored. More than one visiting sequence of nodes exists and in this approach any of the following can be used.

15043726, 13507246, 13570246 and 10537246.

Whereas in the back-to-front visiting sequence, octants farther away from the viewer are visited before the closer ones. An octant visited later in the sequence overwrites the painted region of any octant visited earlier. Frieder et al [85] assert that on the average, the front-to-back approach is more efficient of the two, since if a node is obscured, then the entire subtree rooted on it can be ignored.

There are different paradigms for the construction of an octree for an object. Since, it is not always possible to obtain the spatial occupancy array of the objects, octrees are also constructed from other representations/methods. They are

- volume intersection
- converting other representation to octrees.

In the volume intersection method, different 2D silhouettes of the object are generated; the silhouettes are swept along the viewing direction to obtain the 3D swept volume and these swept volumes are intersected to construct the resulting 3D object representation. Chien et al [86a] present an algorithm for generating the octree of the object from three orthogonal silhouettes. Veenstra and Ahuja [85] describe an algorithm to construct the octree representation of a 3D object from nine silhouettes : three 'face-on' views and six 'edge-on' views of an upright cube, in time linear in the number of nodes in the octree.

A depth (range) image of object is similar to intensity image, except that for each pixel of the image, the depth of the voxel rather than the intensity are stored. Connolly [84] constructs an octree from multiple views of the corresponding range quadtrees. Brunet et al [87] present an extended octree representation for storing free form surfaces.

One can also convert other representation to the octree representation. Yau et al [83] describe an algorithm for the construction of an octree from quadtrees of serial-sections.

The notion of a pointerless structure to store a tree is extended to the octrees too [Gargantini 82c]. As with linear quadtrees, many an operation can be performed using linear octrees - connected component separation [Gargantini et al 82b] and determining the 3D border [Atkinson et al 85b]. Gargantini [82c] presents the other operations that can be done on linear octrees.

## **2.2.2 CELL DECOMPOSITION**

This is generalized form of spatial enumeration in which the disjoint cells are not necessarily cubical or even identical. In this model, the object is partitioned into rectangular parallelepipeds.

### **2.2.2.1 RECTANGULAR PARALLELEPIPED CODING**

Reddy et al [78] have proposed an extension of the point quadtree to the 3D case, wherein the universe is decomposed into rectangular parallelepipeds by planes perpendicular to the X, Y and Z axes. They have also proposed another method of partitioning the universe into rectangular parallelepipeds. But,

here the rectangular parallelepipeds are not necessarily aligned with any axis, as in the above method. A tree structure is used to represent the object. At the root level, the tree has  $N$  branches and the  $i^{\text{th}}$  branch contains the transformation matrix  $T_i$ ,  $1 \leq i \leq N$ . The  $4 \times 4$  matrices  $T_i$  are the transformations that convert the object space point into the local coordinates of its parallelepiped. Each parallelepiped is recursively subdivided into parallelepipeds in the local coordinates of the enclosing parallelepiped. This method facilitates easy display of the objects, but requires the spatial occupancy array as the input. Methods that overcome this short fall are those that are based on the volume intersection technique.

Kim et al [86] have extended the 2D rectangular coding scheme [Kim et al 83] for storing binary images. In the rectangular coding, elements in the set of rectangles which partition the binary image, are represented by the coordinates of

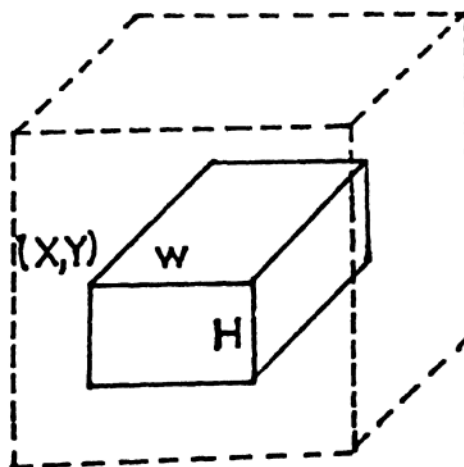


Fig. 2.25 : The rectangle in the front surface is stored as  $(X,Y,W,H)$  and the rectangular parallelepiped is stored as  $(X,Y,1,W,H,N)$ .

the vertex closest to the origin of the image coordinate system, its width and height, i.e.,  $(X,Y,W,H)$ .

The rectangular parallelepiped coding or RP-coding, of an object is the set of rectangular parallelepipeds that partition the object volume. Each rectangular parallelepiped is represented by the  $(X,Y,Z)$  coordinates of the vertex closest to the origin of the coordinate system, its width, height and depth i.e.,  $(X,Y,Z,W,H,D)$ . See Fig. 2.25. The RP-code of an object is constructed from a set of three mutually orthogonal projections (silhouettes) of the object. First, for each of the silhouette, 2D rectangular codes are computed and are swept along the corresponding viewing direction to generate a set of rectangular parallelepipeds (Swept volumes). And then, the swept volumes are intersected to generate the RP-code of the object.

From the 2D rectangular coding, generating the 3D swept volume is a simple matter of rewriting the codes with insertion of '1' (the minimum value of coordinate in the direction of sweeping) and the image size  $N$ . For instance, the swept volume of a rectangle  $(X,Y,W,H)$  of the front silhouette becomes  $(X,Y,1,W,H,N)$ .

Let RP-Top, RP-Front and RP-side denote the swept volumes generated by the top, the front and the side silhouettes respectively. The RP-Code is generated in two steps.

- (i) intersect RP-Front and RP-Top, to generate RP-I
- (ii) intersect RP-Side and RP-I.

```

-----
function GENERATE (RP-Front,RP-Top,RP-I)
/* generates the intermediate swept volume RP-I obtained by intersecting
RP-Front and RP-Top.*/

begin
  for each F in RP-Front do
    for each T in RP-Top do
      if INTERSECT (XF,WF,XT,WT) then
        begin
          XI = Max (XF,XT)
          YI = YF
          ZI = ZT
          WI = min (XF+ WF, XT+ WT) - XI.
          HI = HF
          DI = DT
          add I to RP-I
        end
      end
    end
  end.
-----

```

### Algorithm 2.13

The function INTERSECT checks if two rectangular parallelepipeds intersect in 3D-space.

```

-----
function INTERSECT(XP,WP,XQ,WQ)
begin
  INTERSECT = false
  if (XP ≤ XQ) and (XP+WP > XQ)
    then INTERSECT = true
  if (XP ≥ XQ) and (XP < XQ+WQ)
    then INTERSECT = true
end.
-----

```

### Algorithm 2.14

The final step is to intersect RP-I and RP-side, which is performed by CONSTRUCT.

```

-----
function CONSTRUCT(RP-I,RP-side,RP-R)
/* generates the resulting RP-code, RP-R, by intersecting RP-I and RP-Side */
begin
  for each I in RP-I do
    for each S in RP-Side do
      if INTERSECT( $Y_I, H_I, Y_S, H_S$ ) and INTERSECT( $Z_I, D_I, Z_S, D_S$ )
        begin
           $X_R = X_I$ 
           $Y_R = \max(Y_I, Y_S)$ 
           $Z_R = \max(Z_I, Z_S)$ 
           $W_R = W_I$ 
           $H_R = \min(Y_I + H_I, Y_S + H_S) - Y_R$ 
           $D_R = \min(Z_I + D_I, Z_S + D_S) - Z_R$ 
          add R to RP-R
        end
      end
    end
  end
end
-----

```

### Algorithm 2.15

Kim et al [86] have presented a comparison of RP-codes and octrees. In RP-code, each rectangular parallelepiped is stored as a tuple of six integers and they are bounded by the size of the image. For an 128 x 128 image, if there are  $n$  rectangular parallelepipeds, the space requirement is  $6n$  bytes.

The RP-codes take less storage than octrees because, after each intersection process in RP-coding scheme, the number of volume elements is reduced by merging rectangular parallelepipeds whereas in octrees, the volume element should always be a cube and merging is not possible i.e., one rectangular parallelepiped in RP-code can be an union of cubes, but a node in octree cannot be a union of more than one rectangular parallelepiped in the corresponding RP-code because there is no more scope for merging of rectangular parallelepipeds.

Using RP-codes, transformations are easy to perform and the properties of volume, surface area and moments can be computed easily, as also the generation of different views of the object [Kim et al 86].



### 2.2.2.2 TRANSLATION INVARIANT DATA STRUCTURE

The Translation Invariant Data structure, TID, proposed by Scott et al [86] for representing 2D objects has been extended to 3D by Iyengar and Gadagkar [88]. Since the same name, TID, is used to represent the 2D, as well as the 3D objects, we use 2TID and 3TID to distinguish these two.

In 2TID, the image is broken up into maximal squares, possibly overlapping, and each square is stored by a triple  $(i,j,s)$ , where  $(i,j)$  is the first pixel of the square and  $s$  is the size of the square. More details can be found in Section 2.1.2.9.

The 3TID is constructed from multiple silhouettes of the object. For simple geometric objects (cubes, cones, cylinder etc.) and objects constructed from these primitives, three views are sufficient. The three views considered here are the top, the front and the right side views. The various phases in the construction of the 3TID are

- (i) for each of the three views, the corresponding 2TID is generated.
- (ii) Eliminate the redundant maximal squares that do not provide any useful information.
- (iii) Represent each maximal square of the 2TID by a quadruple,  $(v,i,j,s)$ , where  $v$  is the view and  $i,j$  and  $s$  are as defined above.

The second phase, involving removal of redundant maximal squares needs to be elaborated. The front and right side views correspond to the last column and last row of the top view

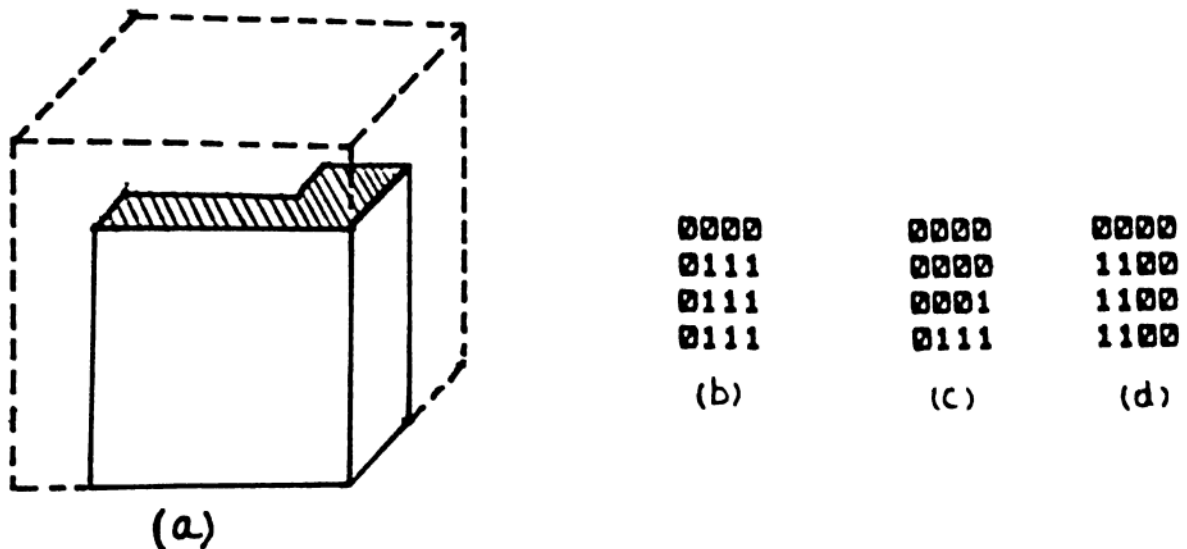


Fig. 2.26 : A 3D object (a), its front (b), top (c) and right side (d) silhouettes.

respectively. Hence, if there are maximal squares of size 1 in the last column (corresponding to the right side) or last row (corresponding to the front) in the top view, they can be deleted. But the other maximal squares in the top view are retained. In case, the front and side views start with the largest maximal square and there are only two, then the following rules are applied.

- (i) If the two maximal squares are in the front view then delete the second maximal square if the right side view is considered and the first, if the left side view is taken.
- (ii) Retain the two if none of the side views are considered.

The authors ascertain that the number of nodes generated in the 3TID is less than or equal to the number of nodes in the corresponding octree. Equality may be attained for simple objects and it is less, when the image contains multiple objects.

Further, for the construction of an octree, the image required is a 3D image as obtained by a CAT scanner, whereas a 3TID needs only multiple 2D views. As the name suggests, translating an object is trivial using 3TID, as is the case with  $90^\circ$ -rotations.

The 3TID is compared to the octree generated from multiple slices of the object [Yau et al 83] and from three mutually orthogonal silhouettes [Chien et al 86a]. The time complexity in case of Yau et al is of  $O(2^{nd})$  where  $n$  and  $d$  are the resolution and the depth of the octree respectively. For the algorithm presented by Chien et al [86a], it is of  $O(S^3 \log S)$ , where  $S$  is the total number of nodes generated in the octree. But, to construct a 3TID, it takes time of  $O(S^2 m \log S)$ , where  $m$  is the total number of maximal squares. The time complexity for this method is much less than that of the other two because  $m \ll S$ .

### 2.2.3 CONSTRUCTIVE SOLID GEOMETRY

The Constructive Solid Geometry, CSG for short, [Voelcker et al 77] is the method of representing objects as a composition of primitive solids. Here, the object is stored as a tree, in which internal nodes store the operations and the leaves, the primitives. See Fig. 2.27.

At the lowest level are the primitive solids, which are the intersection of closed half-spaces defined by  $F(X,Y,Z) \geq 0$ , where  $F$  is well-behaved, for instance, analytic. Generally, primitives are objects like arbitrarily scaled rectangular parallelepipeds, cylinders, cones and spheres of arbitrary radii. They may be positioned arbitrarily in space.

Some of the point-set topology terms used in CSG are

defined below [Simmons 63]. Consider a topological space  $(W, T)$ .

The closure of a subset  $x$ , denoted by  $kx$ , is the union of  $x$  with the set of all its limit points.

The interior of a subset  $x$ , denoted by  $ix$ , is the set of all the interior points of  $x$ .

The subset is regular if  $x = kix$  i.e., if  $x$  is the closure of its interior.

In the CSG scheme, to avoid dangling edges when set operations are performed, the regularized union, intersection, difference and complement defined below are considered and are denoted by superscripting them with a '\*'.

$$X \cup^* Y = ki (X \cup Y)$$

$$X \cap^* Y = ki (X \cap Y)$$

$$X -^* Y = ki (X - Y)$$

$$X^{C*} = ki (X^C)$$

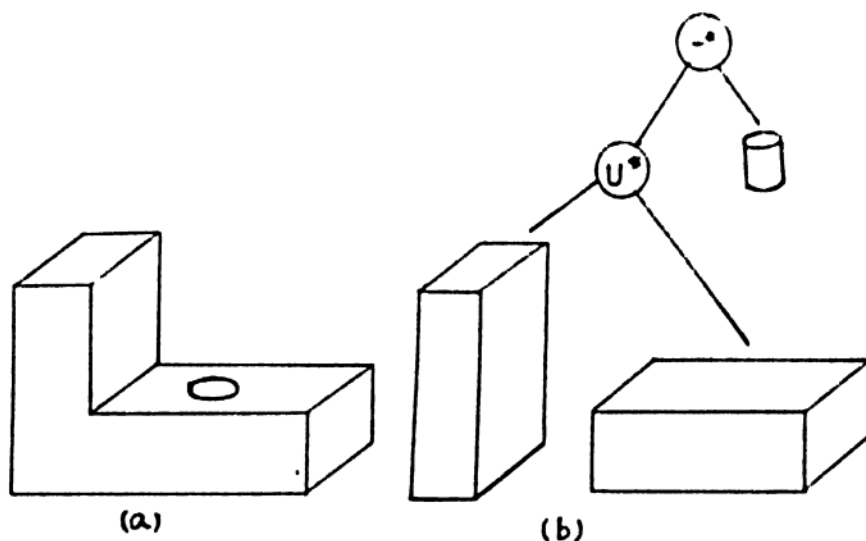


Fig. 2.27 : A 3D object (a), and its CSG representation (b). In the above tree, primitives are stored in the leaf nodes and the regular operators in the internal nodes.

A CSG representation is a Backus-Naur Form expression involving primitives and set operations (intersection, union and difference) for combination and motion.

```
<CSG rep > = < primitive solid> |
              MOVE <CSG rep> BY <motion parameters> |
              <CSG rep><operator><CSG rep>
```

#### 2.2.4 GENERALIZED CYLINDERS

When a 2D object is swept along a 3D space curve or rotated along an axis, different swept volumes are generated. When a 2D set is translated along a line, it is called a translational sweep. By rotating the 2D set around an axis, a rotational sweep is obtained. In the three dimensional sweeps, volumes, rather than 2D sets are swept. In a general sweep, a 2D set or a volume is swept along an arbitrary space curve and the set may vary parametrically along the curve [Soroka 79]. The general sweeps are called Generalized Cylinders, GC. (also called generalized cones).

A GC is a solid whose axis is a 3D space curve. At any point on the axis, a closed cross-section is defined. Usually, it is easier to think of a GC as a space curve with a cross sectional point set function, both parametrized by the arc length along the space curve. For a GC, there exist infinitely many axis-cross section pairs that define it.

A coordinate system that depicts the local behavior of the GC axis space curve is the Frenet Frame, defined at each point on the GC axis. The origin of the frame is the point on the axis itself and the three orthogonal vectors are given by

(  $\xi, \nu, \zeta$  ) where

$\xi$  = unit vector tangent axis

$\nu$  = unit vector direction of the center of the curvature of axis normal curve, and

$\zeta$  = unit vector direction of center of torsion of axis .

The Frenet frame gives good information about the GC axis, but has certain drawbacks. It is not well-defined when the curvature of the GC axis is zero. Further it may not reflect known underlying physical principles that generate the cross sections. For instance, consider the case of bolt threads. They can be described by a single cross section that stays fixed in a coordinate system that rotates as it moves along the straight axis of the bolt.

The cross section is defined as a point set in  $\nu$ - $\zeta$  plane, using inequalities expressed in the  $\nu$ - $\zeta$  coordinate system. It can also be described by the cross section boundary, parametrized by another parameter  $r$ . Let this curve be given by  $(x(r,s), y(r,s))$ . The dependence on  $s$  shows that the cross sectional shapes can vary along the axis. The above expression, in world coordinates, is transformed to the local coordinates by

$$B(r,s) = a(s) + x(r,s)\nu(s) + y(r,s)\zeta(s)$$

where  $B$  is the GC boundary and  $a$  is the axis.

The GC facilitates easy computation of many parameters of the solid.

- the perpendicular to a cross section, from  $(x(r,s), y(r,s))$

is given by

$$\frac{dy}{ds}\nu - \frac{dx}{ds}\zeta$$

- the area of a cross section is given by

$$\int_0^P (x \frac{dy}{ds} - y \frac{dx}{ds}) ds$$

where P is the perimeter.

- the volume is given by the integral of the area as a function of the axis parameter multiplied by the incremental path length of the GC axis.

$$\int_0^L \text{area}(s) ds$$

where L is the length of the GC axis.

## 2.2.5 BOUNDARY REPRESENTATIONS

In this scheme, the objects are represented by their enclosing surfaces, such as planar, quadric surfaces, patches etc. The representation of a space curve and a surface in three dimensional space are distinguished.

### 2.2.5.1 SPACE CURVES

A digital space curve is defined as a connected set of voxels all but two of which have exactly two neighbors in the set and the end voxels have exactly one neighbor each. The 2D chain coding is extended to the third dimension.

In 3D space, a voxel can have 6, 18 or 26 possible directions to the neighboring voxel depending on whether the adjacency considered is face, face and edge or face, edge and vertex. These are also called 1-, 2- and 3-neighbors respectively [Srihari 81]. See Fig. 2.28.

Thus each possible direction can be uniquely described by 3 bits for 1-connected space curves, and 5 bits for 2- or 3-connected space curves. For a voxel at (x,y,z), the 26

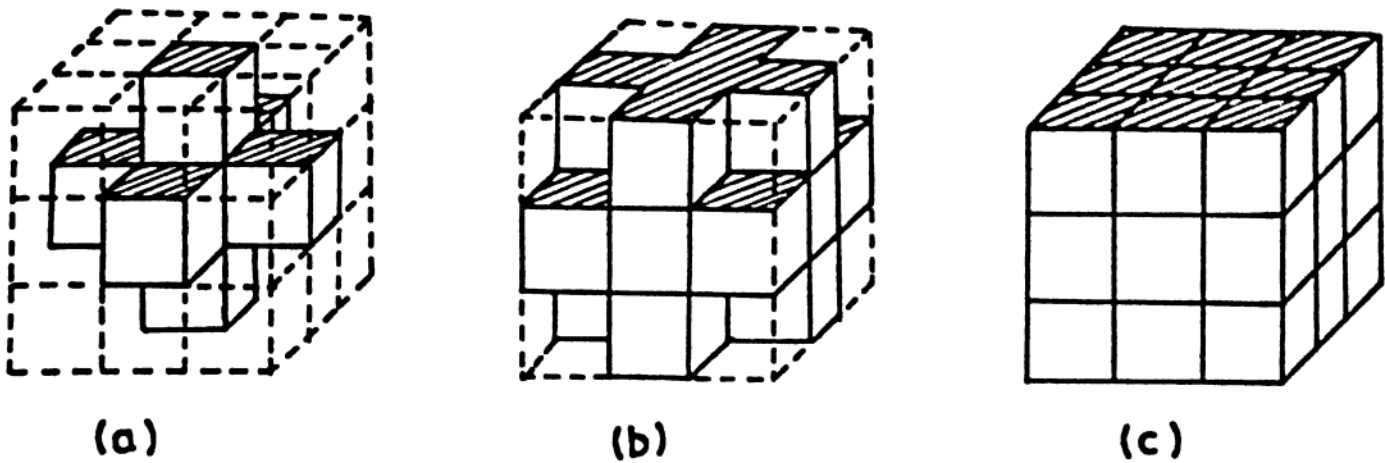


Fig. 2.28 : 1-neighbors (a), 2-neighbors (b) and 3-neighbors (c) of a voxel in 3D space.

neighboring voxels are given by  $(x \text{ or } x \pm 1, y \text{ or } y \pm 1, z \text{ or } z \pm 1)$  and hence, a space curve can be stored as a word over the alphabet  $\{ -1, 0, 1 \}$  and the coordinates of the starting voxel.

#### 2.2.5.2 SURFACES

The representation of surfaces is vital for the display of the shaded surfaces of an object on the screen. The surface of an object  $S$ , may be defined as the voxel faces that are at the interface of  $S$  and  $S^C$  (the set complement of  $S$  in the universe). If  $S$  is 1- connected, then  $S^C$  is 3- connected and vice-versa. The surface of an object  $S$  is formally defined as the set of voxel pairs

$$Y(S) = \{ (u,v) \mid u \in S, v \in S^C, u \text{ is a face neighbor of } v \}$$

The surface of an object can be represented by specifying the set of constituent faces or indirectly by means of border voxels, medial axes and graphs (where vertices correspond to faces and edges to touching faces).



The border of  $S$ ,  $B(S)$ , is defined by

$$B(S) = \{ v \mid v \in S \text{ and } N_3(v) \cap S^c \neq \emptyset \}$$

where  $N_3(v)$  denotes the set of 3-neighbors of  $v$ . By the very definition of  $Y(s)$ , it follows that the surface of  $S$  is uniquely specified by  $B(S)$ .

To determine if a voxel  $v$  belongs to  $B(S)$  is to check if any voxel of  $N_3(v)$  belongs to  $S^c$  i.e., if  $\{ v^i \}$  are elements of  $N_3(v)$ , then

$$v \in B(S) \Leftrightarrow \chi(v) \left( \prod_i (v^i) \right) = 1$$

where  $\chi$  is the characteristic function of the object  $S$ .

## 2.3 CONCLUSIONS

A wide spectrum of data structures for the representation of objects has been presented. As mentioned earlier, no single data structure admits all the desirable properties one would ask for. At times, it would be convenient to convert one representation scheme to another for fast query processing and certain other operations. Among the 2D object representation schemes, the following algorithms are proposed for converting one representation to another : boundary to skeleton and vice-versa [Rosenfeld et al 66], boundary to bintrees [Diehl 88 ], chain codes to quadrees [Samet 80b], quadrees to chain codes [Dyer et al 80], arrays to quadrees [Samet 80a], rasters to quadrees [Samet 81b], quadrees to rasters [Samet 84b], quadrees to QMATs [Samet 83] and reverse [Samet 85c], quadrees to edge quadrees [Ayala et al 85], normalizing quadrees [Chien et al 84], raster to linear quadrees [Gargantini 82a, Abel et al 83, Samet 85b, Shaffer et al 87] and treecode to leafcode [Oliver et al 83b].

More details on 2D object representation schemes can be found in [Shapiro 79, Rosenfeld et al 82, Ballard et al 82, Overmars 83, Samet 84a, Samet et al 85].

The conversion algorithms for 3D objects are presented by different workers. Some of them are CSG to octree [Lee et al 82, Samet and Tamminen 85], boundary representation to octrees [Tamminen et al 84 and Tang et al 88] and cylinder representation to octree [Requicha 80] and boundary to CSG [Juan 88].

The 3D object representation can also be generated from the cross sections of the object, as well as from different silhouettes. Based on the paradigm of volume intersection, octrees can be generated from different types of input data. For example, octrees can be generated from quadtrees of serial sections [Kim et al 86], quadtrees of silhouettes [Chien et al 86b and Veenstra et al 86] and depth image quadtrees [Connolly 84].

In between 2D and 3D objects, there is another class of objects, known as  $2\frac{1}{2}$  D objects. The two dimensional manifolds in three dimensional space are defined as  $2\frac{1}{2}$  D objects. A two dimensional manifold may be thought of as a topological space with the property that every open set is diffeomorphic to an open set in  $R^2$ . i.e., locally, it behaves like  $R^2$ . The surfaces and boundaries of 3D objects are  $2\frac{1}{2}$  D objects.

The judicious selection of proper data structure is difficult because of the plethora of the available data structures. Some of the aspects to be considered in the choice of an appropriate data structure are

- (i) space : the amount of space required to store the data

structure, as well as the space required for any temporary data/data structure is to be considered.

- (ii) time : the construction of the data structure that represents the object in real-time applications needs to be very prompt.
- (iii) fast query processing : one should be able to answer the queries in a reasonable time. When the query processing is the top priority, depending on the type of queries, one can sacrifice the high time requirement to construct the data structure.
- (iv) nature of data : Depending on whether the data is static or dynamic, i.e., some data structures allow easy modification of the object.
- v) display : a good data structure should enable one to reconstruct the object from the representation exactly, without any aberrations and approximations.
- vi) translation invariance : in pattern recognition tasks, it is desirable to have data structures that are invariant to translations. The MAT, the TID, the RP-codes and the chain codes are translation invariant.

### 3.1 EDGE k-d TREE

Several methods to store spatial information were devised and the major schemes are presented in Chapter 2. The prominent among them are the raster and vector representations, chain codes, pyramids and the numerous variants of the ubiquitous quadtree. In this Chapter, a new data structure is presented, namely the edge k-dimensional binary search tree or edge k-d tree, for short. This is a variant of the conventional k-d tree [Bentley 75] ( cf. 2.1.0.2) and the edge quadtree [Ayala et al 85] ( cf. 2.1.1.4) but it has the additional features of being balanced, translation and scaling invariant.

When the object is represented in the form of a tree, the nature of the tree is important because all further operations and information retrieval depend on it. In particular, the balanced nature and the height of the tree play a crucial part. The time to retrieve information from the tree or the traversal of the tree directly depend on these. The shorter the height, the less is the time it takes to access any node. Moreover, if the tree is balanced and information is stored systematically, the binary search, with a time complexity of  $O(\log N)$ , can be used for information retrieval. Among the data structures that are presented in Chapter 2, the k-d tree is balanced. But none of the important tree structures is balanced.

---

<sup>\*</sup>The main results of this chapter were presented in a National Conference. See [Lavakusha et al 86]

Not many representation schemes are invariant to translation and scaling. A slight change in the position or size of the object, may necessitate drastic reorganization of the representation. In many applications, a translation invariant data structure is necessary. The TID, MAT, polylines and chain-codes are invariant to translation and scaling. In fact, the normalized derivative of the chain-code (Sec. 2.1.1.2) is used as a shape descriptor [Pavlidis 78]. Though the quadtree has many advantages, it is not translation invariant. In Fig. 2.14, how the quadtree structure changes when the object is translated by one unit along the X-axis is shown. Li et al [82] have proposed a normal form of quadtree that is translation invariant. Chien et al [84] presented the normalized quadtree that is invariant to translation, scaling and rotation. But the space and time complexities are of  $O(2^{2l})$  and  $O(1.2^{2l})$  respectively for finding the minimal cost quadtree in terms of the number of nodes, by moving an image lying between squares  $2^{l-1}$  and  $2^l$  in a square of size  $2^{l+1}$ . Thus though there are several data structures for 2D region representation, no single scheme provides for the two desirable properties such as balanced tree structure with translation invariance. This motivated us to develop an alternative representation, the edge k-d tree, which is balanced and translation invariant.

### 3.2 CONSTRUCTION OF THE TREE

Now the underlying idea of constructing the edge k-d tree for any simple polygon  $P$  of  $N$  edges is described. This is a divide-and-conquer algorithm akin to the quicksort algorithm

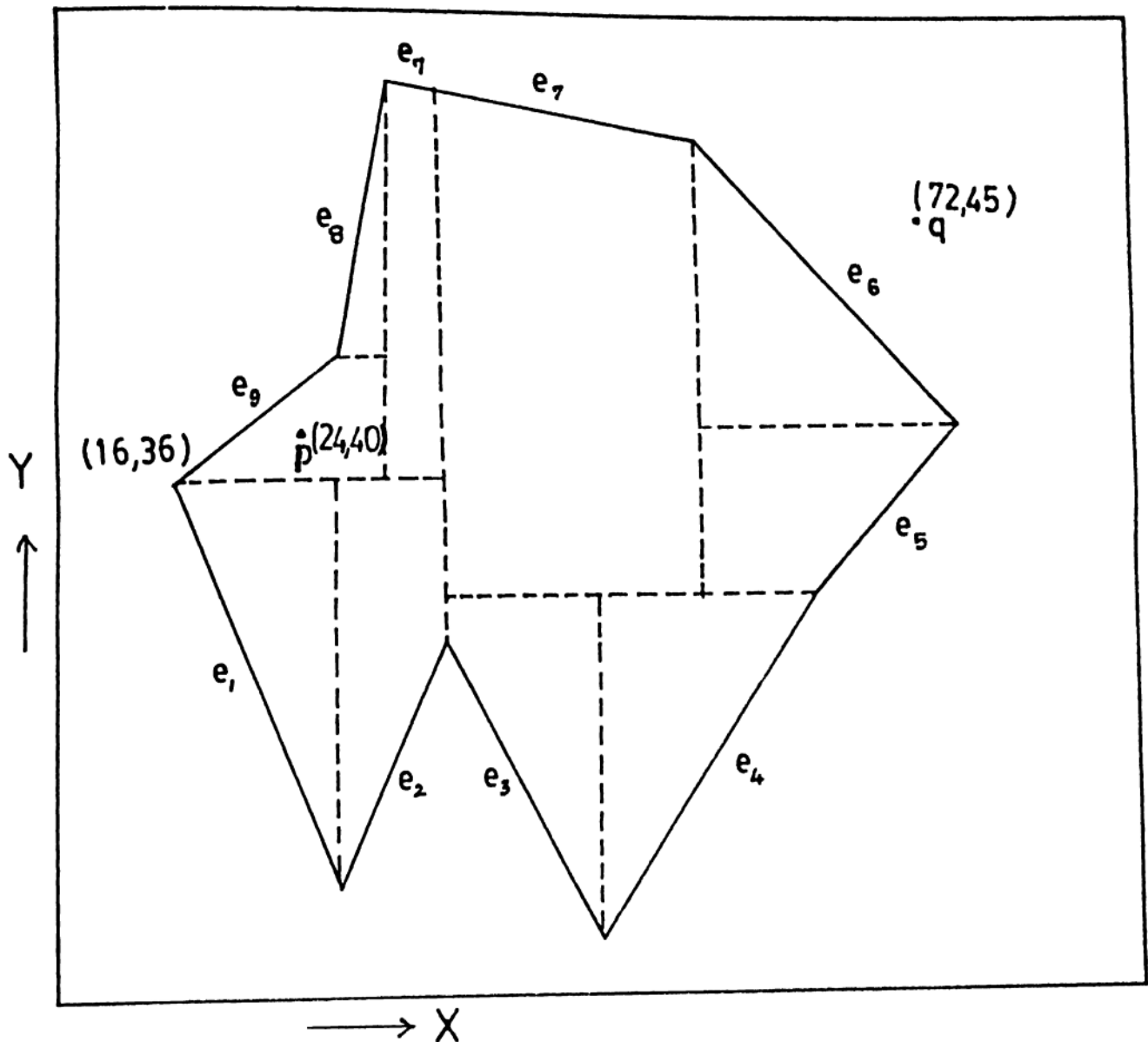


FIG. 2.1 : A non-convex polygon. The vertices are given in clockwise direction and edges are denoted by  $e_1$  s.

[Hoare 61]. It is assumed here that the vertices of  $P$  are arranged in clockwise order. During the construction, the original list of vertices is recursively divided into smaller lists. We call the list that is to be divided as the object list. Initially the median of the  $X$ -coordinate of the vertices in the object list is computed. On this, the object list is partitioned into two sublists. For the first sublist, the  $X$ -coordinates of

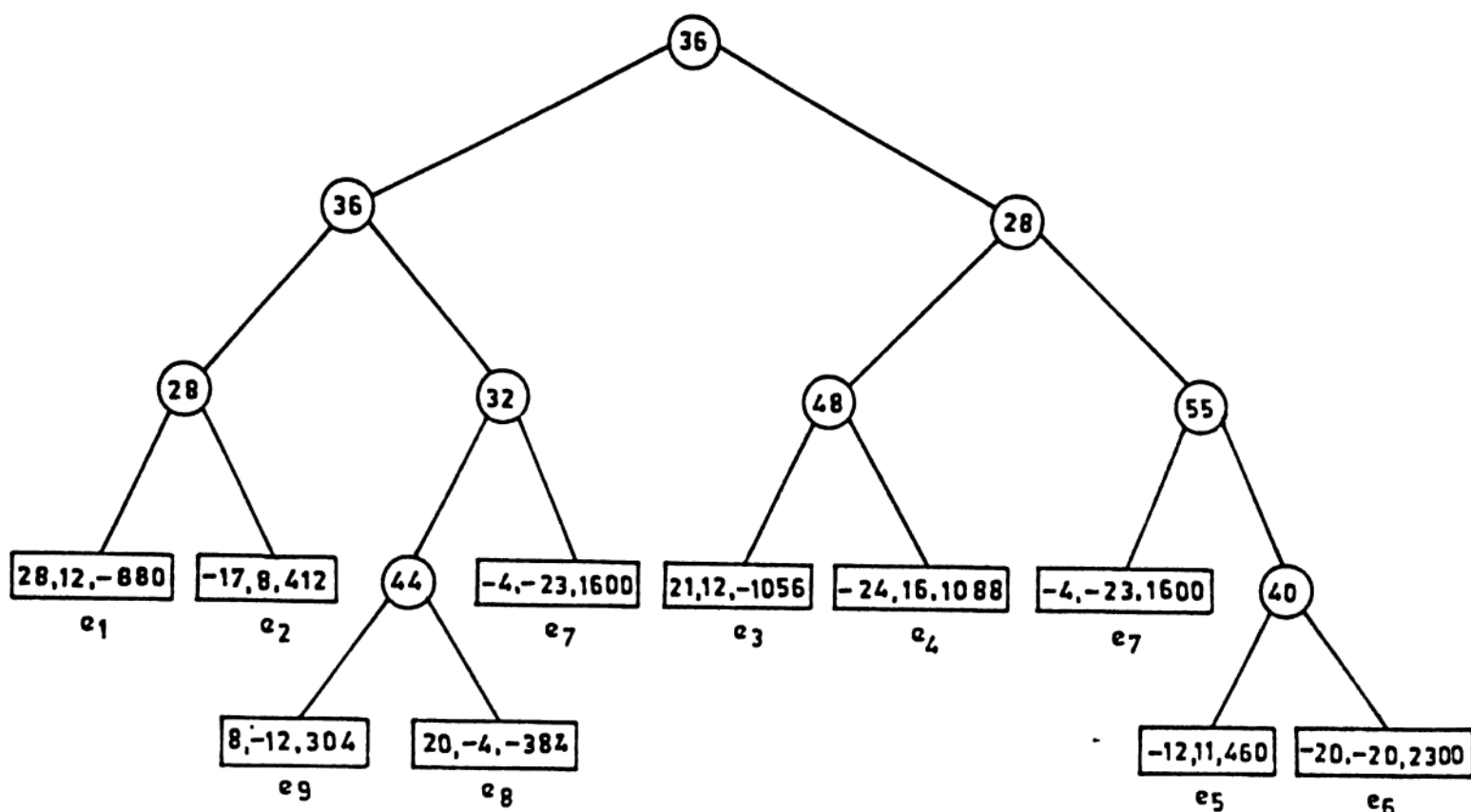


FIG. 3.2 : The edge 1-d tree representation of the polygon in Fig. 3.1. The  $e_i$ 's below the leaves indicate the edges they represent.

the vertices is less than the median and for the other sublist, it is greater. Now each of the two sublists will be the object lists and the above process is repeated for each of the object lists, but this time the median is computed for the Y-coordinates. This recursive process of computing the median alternately on X and Y-coordinates and then partitioning the object list is repeated while the number of vertices in any object list is greater than three.

An X-cut is made, when the object list is partitioned into two sublists based on the median of the X-coordinates of the vertices. The vertex at which the cut is made is called object vertex. Similarly for Y-cut. In short, an X-cut is made on the

original object list. On the resulting two object lists, Y-cuts are made and on the resulting four object lists, X-cuts are made and so on while the cardinality of the object list is four or more.

But when the cardinality of the object list is three or less, the vertex that is farthest from the previous cut is chosen for the purpose of partitioning. Because of the cut being made at the farthest point, rather than at the median, the 'balanced' nature of the binary tree is lost. This may result in the tree having a height of  $\lceil \log N \rceil + 1$ , but still the height of the tree is  $O(\log N)$ . The structure of a node in this binary tree is

|       |      |      |      |   |   |
|-------|------|------|------|---|---|
| coord | lptr | rptr | fptr | b | c |
|-------|------|------|------|---|---|

The 'coord' field represents the X (Y) coordinate where the X-cut (Y-cut) is made. The next three fields are the pointers to the left and right children of the node and the father of the node. For the internal nodes, the fields b and c are blank. The coefficients a, b and c of the line equation  $aX+bY+c=0$  are stored in the first, the fifth and the sixth fields respectively.

The relation between the list of vertices and the edge k-d tree is described below. The X-coordinate of the vertex where the X-cut is made in the original object list is stored in the 'coord' field of the root of the tree. This results in two object lists. The first sublist for which the X-coordinate is less than the X-coordinate of the object vertex, corresponds to the left branch of the tree and the other sublist corresponds to the right branch. In general, for any internal node, the left (right)



branch corresponds to the sublist created by that X-cut (Y-cut) for which the X (Y) coordinate of the object vertex is less (greater) than the X (Y) coordinate of the object vertex. The edge equations are stored in the leaf nodes, such that for points inside the polygon, the value obtained by substituting the points' coordinates in the edge equation is positive. The polygon is considered to be closed (in the sense of point set topology) i.e., the boundary of the polygon is part of the polygon.

A polygon and its edge k-d tree are shown in Fig. 3.1 and Fig. 3.2 respectively. Here there are nine vertices, given in a clockwise direction. When the array is sorted on X-coordinate, the median vertex is (36,25) and the left and right subarrays are { (16,36), (28,44), (28,8), (32,64) } and { (48,4), (55,60), (64,28), (75,40) } respectively. In the root of the tree, the X-coordinate of the median vertex 36 is stored. Each of the two subarrays is sorted on Y-coordinate. We follow the action on the left subarray. Sorted on Y-coordinate, the left subarray becomes { (28,8), (16,36), (28,44), (32,64) }. The median vertex's Y-coordinate, i.e., 36 is stored in the left son of the root. Now, the left subarray is just one vertex (28,8) and the right subarray is { (28,44), (32,64) }. The X-coordinate 28 is stored in the left son. And for the right branch, there are only two vertices. Here, the vertex farther from the previous cut is taken i.e., the vertex (32,64) is farther of the two vertices from the cut  $Y = 40$ . Hence an X-cut is made at (32,64) and 32 is stored in the right son. The Y-coordinate of the remaining vertex is stored in the left son.

For generating the line equations, the vertex is searched for

in the original array and its neighbors are located and the equation is computed using the formula

$$\frac{x-x_1}{x_1-x_2} = \frac{y-y_1}{y_1-y_2}$$

if the given vertices are  $(x_1, y_1)$  and  $(x_2, y_2)$ . After the coefficients of the equation  $a, b, c$  are computed, they are multiplied by  $-1$ , if necessary, so that for points inside the polygon  $aX + bY + c > 0$ .

### 3.3 THE ALGORITHM

The input to the algorithm is the number of points  $N$ , the coordinates of the vertices given in either clockwise or counterclockwise direction and a reference point  $r$ . We need the point  $r$  to determine the signs of the coefficients in the edge equations. The algorithm generates the edge  $k$ -d tree. We use two arrays of size  $N \times 2$ , namely `points` and `dummy`, to store the vertices. The sorting of vertices subsequently is done on the array `dummy`. Each node in the edge  $k$ -d tree has fields and a record structure `treenode` is used.

```
struct treenode (
    int coord;
    pointer lptr, rptr, fptr;
    int b, c;
)
```

The algorithm in a C-like pseudo code is presented in Alg.3.1 - 3.3.

```
.....
main()
begin
    read N and the vertices into the array points;
    copy points into dummy;
```

```

t = getnode();
dir = 0;
sort the array dummy on X-coordinate and let the median be i;
t.coord = dummy[i,0];
t1 = getnode();
t2 = getnode();

construct(1, i-1, t1, 1);
construct(i+1, N, t2, 1);

write the tree in preorder in a file;
end

```

---

### Algorithm 3.1

getnode creates a record of the type treenode and returns the address of the new record. The X and Y coordinate of the vertex  $j$ ,  $1 \leq j \leq N$ , is stored in `points[j,0]` and `points[j,1]` respectively. Same is the case with `dummy`.

---

```

construct (f, l, tnode, dir)
/* a recursive function to build the edge k-d tree. f and l are the first and
last indices in the subarray of the array dummy. dir determines whether the
next cut is to be an x-cut or a Y-cut */

dif = l-f;
case (dif) of
  begin
    0 : /* there is only one point */
      t1 = getnode();
      t2 = getnode();

      for the vertex (dummy[l,0], dummy[l,1]) find the
      previous and the next vertices from points.

      tnode.coord = dummy[l,dir];

      i = search ( dummy[l,0], dummy[l,1]);
      if ( i = N ) then pos = 0;
        else pos = i+1;

      if (points[pos,dir] > points[i,dir] )
        equation (i-1, i, tnode.lptr);
        equation (i, i+1, tnode.rptr);
      else
        equation (i, i+1, tnode.lptr);
        equation (i-1, i, tnode.rptr);
      endif;
  end;

```

Algorithm 3.1 ( contd.)

```
1 : /* there are two points. It takes the farthest point from the
      previous cut and introduces a cut there */
```

```
prev_cut = tnode.fptr.coord;
cdir = 1 - dir;
```

```
fdif = | prev_cut - dummy[f,dir] |;
ldif = | prev_cut - dummy[l,dir] |;
```

```
if ( ldif > fdif ) then pos = l
                      else pos = f;
```

```
tnode.coord = dummy[pos,dir];
k = search(dummy[pos,0] dummy[pos,1]);
```

```
depending on whether the cut is made at f or l,
construct is invoked for the left or right son and an equation
is created at the other son;
```

```
2 : /* there are three vertices. This module introduces a cut at the
      farthest vertex from the previous cut. construct is called once
      and the equation once. */
```

```
prev_cut = tnode.fptr.coord;
```

```
let pos be the vertex ( one of f or l ) that is farthest
from prev_cut along the direction dir;
```

```
k = search (dummy[pos,0] dummy[pos,1] );
```

```
depending where the cut is made, i.e., either at f or
l, construct is called twice or a construct and an
equation are called;
```

```
default : /* there are more than three points. This module sorts the vertices
along dir, takes the median, make the cut there and invokes
construct for the resulting two subarrays. */
```

```
sort (f, l ,dir);
dif = (f+l)/2;
tnode.coord = dummy[dif,dir];
```

```
construct(f,dif-1,cdir,tnode.lptr);
construct(dif+1,l,cdir,tnode.rptr);
```

```
end;
```

```
end.
```

### Algorithm 3.2

The function sort(f, l, dir), sorts the vertices in the

array dummy only between the indices  $f$  and  $l$ , with the primary key on  $dir$ . The function  $search(x,y)$  searches for the location of the vertex  $(x,y)$  in the array  $points$  sequentially.

```

.....
equation(i, j, tnode)
begin
    x1 = points[i,0];
    x2 = points[i,1];
    y1 = points[j,0];
    y2 = points[j,1];

    coord = y1 - y2;
    b = x2 - x1;
    c = x1 * (-a) - y1 * b;

    if (r1*a + r2*b + c < 0) then sign = -1
                                   else sign = 1;

    tnode.coord = sign*coord;
    tnode.b = b*sign;
    tnode.c = c*sign;
end
.....

```

### Algorithm 3.3

#### 3.4 ANALYSIS

The algorithm makes use of the functions  $main$ ,  $construct$ ,  $sort$ ,  $equation$  and  $search$ . The time complexity of  $sort$  is  $O(n \log n)$  if there are  $n$  records, by Quicksort. The worst case complexity of  $search$  is  $O(N)$ . The time complexity of  $equation$  is  $O(1)$ .

The complexity of the function  $construct(l,f,tnode,dir)$  depends on the length of subarray of the vertices that it has to organize, i.e., on  $l-f$ . Two cases arise here, i.e., when  $l-f \leq 3$  and otherwise.

In the first case, when  $l-f \leq 3$ ,  $search$  is called once and the complexity of the search for a particular vertex in an array of size  $N$  is  $O(N)$ .

In the other case, when  $1-f > 3$ , the complexity depends on the sort and the complexity of sort is of  $O(N \log N)$ , when there are  $N$  vertices.

Hence, the complexity of construct is  $\max(O(N), O(N \log N))$ , i.e.,  $O(N \log N)$ .

The initial array of size  $N$  is sorted and construct is invoked once for each of the two subarrays of size  $N/2$  and four times for subarrays of size  $N/4$  and so on. i.e., construct is invoked  $2^i$  times for subarrays of size  $N/2^i$ . Hence the complexity of the function main and hence of the algorithm is

$$\begin{aligned}
 & (N \log N) + 2\left(\frac{N}{2} \log \frac{N}{2}\right) + 4\left(\frac{N}{4} \log \frac{N}{4}\right) + \dots + 2^{\log N} \\
 &= \sum_{i=0}^{\log N} 2^i \left(\frac{N}{2^i} \log \frac{N}{2^i}\right) \\
 &= \sum_{i=0}^{\log N} N(\log N - \log 2^i) \\
 &= N \log^2 N - \sum_{i=0}^{\log N} N \cdot i \\
 &\approx N \log^2 N - \frac{(N \log^2 N)}{2} \\
 &\approx \frac{(N \log^2 N)}{2}
 \end{aligned}$$

Hence the time complexity of constructing the edge  $k$ -d tree is  $O(N \log^2 N)$ .

### 3.5 OPERATIONS ON EDGE $k$ -d TREES

In this section, the operations of answering the point membership problem, the translation of the polygon by a vector  $(t, s)$  and the scaling of the polygon by a constant  $s$  are

addressed using the edge k-d tree.

### 3.5.1 POINT MEMBERSHIP

The 'point-in-polygon' problem i.e., given a polygon  $P$  with  $N$  sides, and a point  $p$ , to determine whether  $p$  is inside  $P$ , can be answered in  $O(\log N)$  time using the edge k-d tree.  $O(\log N)$  is the best known bound for any simple polygon [Preparata 88]. The point-in-polygon problem is also referred to as the point membership problem. Given a point,  $p = (p_1, p_2)$  and the edge k-d tree of the polygon  $P$ , if the point  $p$  is inside  $P$  is checked. The algorithm for answering this problem is a simple tree traversal method. We compare  $p_1$ , the X-coordinate of  $p$  with the 'coord' of the root and take the left branch if  $p_1$  is less than the coord of the root. Else, the right branch. Next, we compare  $p_2$ , the Y-coordinate of  $p$ , with the coord of the node and depending on

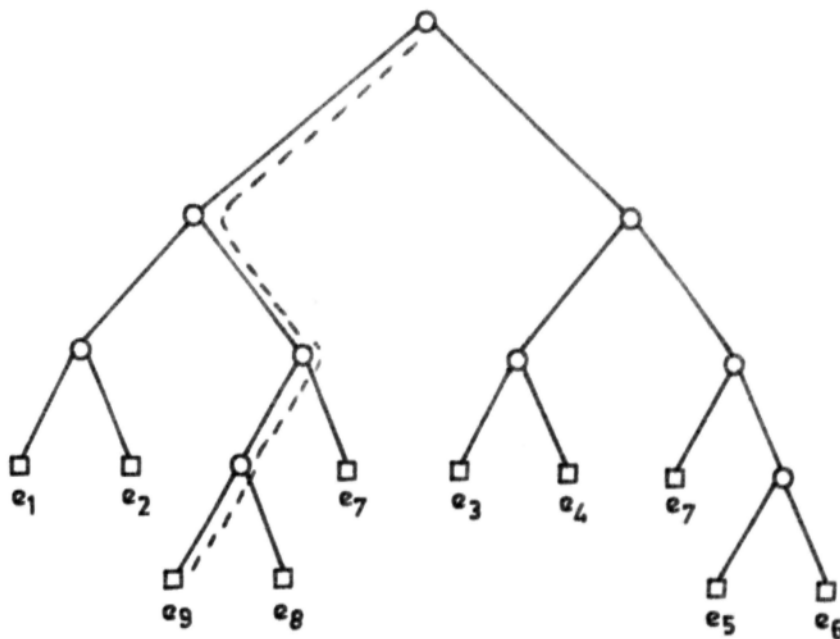


Fig. 3.3 : To determine the membership of the point  $p=(24, 40)$ , the nodes visited are shown. The edge equation is  $8X - 12Y + 304 = 0$ .

whether  $p_2$  is less or greater than the coord field, we take the left or the right branch. Thus, alternating between  $p_1$  and  $p_2$ , we keep comparing with the coord field and take the appropriate branch for the next comparison. On coming to the leaf, we have an edge equation. By substituting  $(p_1, p_2)$  in the edge equation, if the result is greater than 0, then  $p$  is inside  $P$ , else outside. The algorithm is given in Alg. 3.4.

```

-----
function member (ptr,p)
/* ptr is pointer to the root of the edge k-d tree and p=(p1,p2) */

begin

    i=1;

    while ( ptr is not leaf) do

        begin
            if pi < ptr.coord
            then ptr = ptr.lptr
            else ptr = ptr.rptr;
            i = 3-i;
        end;

    /* this while loop determines the block containing the point p. From this
       leaf we can access the corresponding (a,b,c) of the edge, this leaf
       represents */

    k = a * p1 + b * p2 + c;

    if k ≥ 0 then print "Inside"
                else print "Outside";

end.
-----

```

#### Algorithm 3.4

From the algorithm, it is easily seen that, by traversing the binary tree of height  $O(\log N)$ , the point membership problem is answered. Hence, the complexity (best, worst and average) is of  $O(\log N)$  and this is the best known bound for any general simple polygon [Preparata 88].



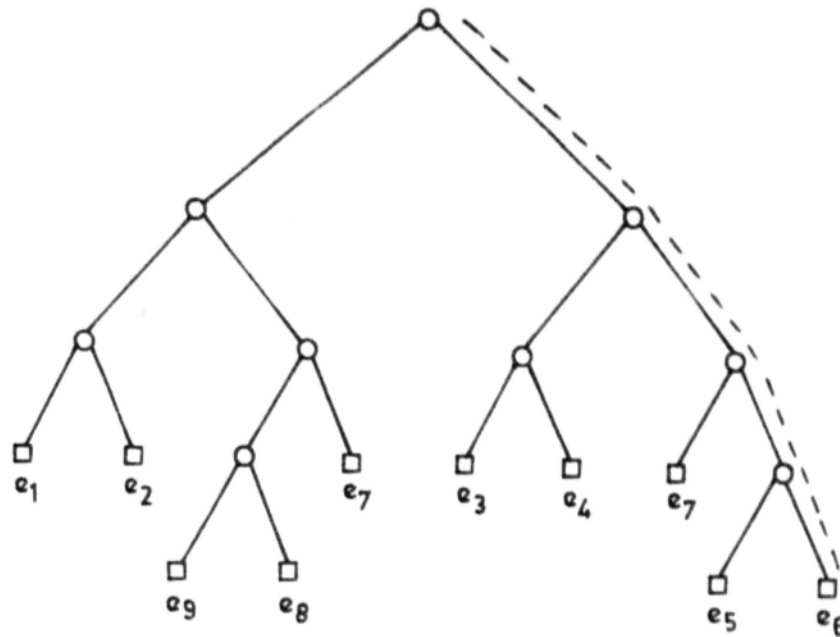


Fig. 3.4 : To answer the membership of the point,  $q = (72, 45)$ , the traversed path is depicted. On substituting  $q$  in the expression  $-20X - 20Y + 2300$ , the result is negative.

### 3.5.2 SCALING

If the polygon  $P$  is scaled by a factor  $s$ , i.e., a vertex  $v = (v_1, v_2)$  becomes  $(sv_1, sv_2)$ , then the edge k-d tree for the scaled polygon can be directly obtained from  $P$ . For this, in the internal nodes of the edge k-d tree, the coord field is multiplied by  $s$  and in the leaf nodes, the field 'c' in the edge equation  $aX + bY + c = 0$  is multiplied by  $s$  i.e.,  $c$  is replaced by  $sc$ . The formal algorithm is presented in Alg. 3.5.

```

-----
function scale (ptr,s)
/* ptr is the pointer to the root of the edge k-d tree and s is the scaling
   factor */

begin
  for (each internal node )do
    coord = coord * s

    for ( each leaf node) do
      c = c * s;
end
-----

```

### Algorithm 3.5

The algorithm is a binary tree traversal and hence can be performed in  $O(m)$  time, where  $m$  is number of the nodes.

#### 3.5.2 TRANSLATION

The algorithm for the translation of the polygon  $P$  represented by its edge k-d tree is also direct. When it is required to translate the polygon by a vector  $(t,s)$ , we traverse the binary tree and make the necessary changes. For the internal nodes, the coord field is updated by adding  $t$  or  $s$  depending on whether it represents an X-cut or a Y-cut. For the leaf nodes, the field  $c$  is replaced by  $'-a * s -b * t -c'$ . We formally present the algorithm in Alg. 3.6.

```

-----
function translate (ptr,t,s)
begin

  for ( each internal node) do
    if ( level of node is even)
      then coord = coord + t
      else coord = coord + s;

    for ( each leaf node ) do
      c =m -a * s -b * t -c;

end.
-----

```

### Algorithm 3.6

The level of the root is 0 and the level of a node in the tree is 1 more than the level of its father. Since this algorithm is a tree traversal, the time complexity is of  $O(m)$ , where  $m$  is the number of nodes.

### 3.6 CONCLUSIONS

It is observed in the previous sections that the edge k-d tree representation facilitates efficient method for solving the point membership problem, translation and scaling operations. In the conventional vector representation, the operations of translation and scaling become trivial, but the point membership problem can not be answered easily. Similarly, though the construction of a region quadtree is relatively easy, this structure is sensitive to scaling and translation. Moreover, the region quadtree is not balanced and hence its size is not a function of the number of vertices of the polygon.

Thus the edge k-d tree can be the most suitable data structure for region representation in situations like automated cartography, GIS and robotics where the point membership, scaling and translation are required frequently. In the polygon partitioned by the X- and Y- cuts in Fig. 3.5, the X-cut at the vertex is singular. Should the edge k-d tree be constructed for the polygon, how should the edges  $e_1$  and  $e_2$  be stored? In an ideal situation, the edges  $e_1$  and  $e_2$ , were to lie on either side of the cut for answering point membership query. But here, since  $e_1$  and  $e_2$  lie on the same side, the present algorithm does not work. Thus, the proposed method of construction of the edge k-d tree works successfully for class of polygons which includes the

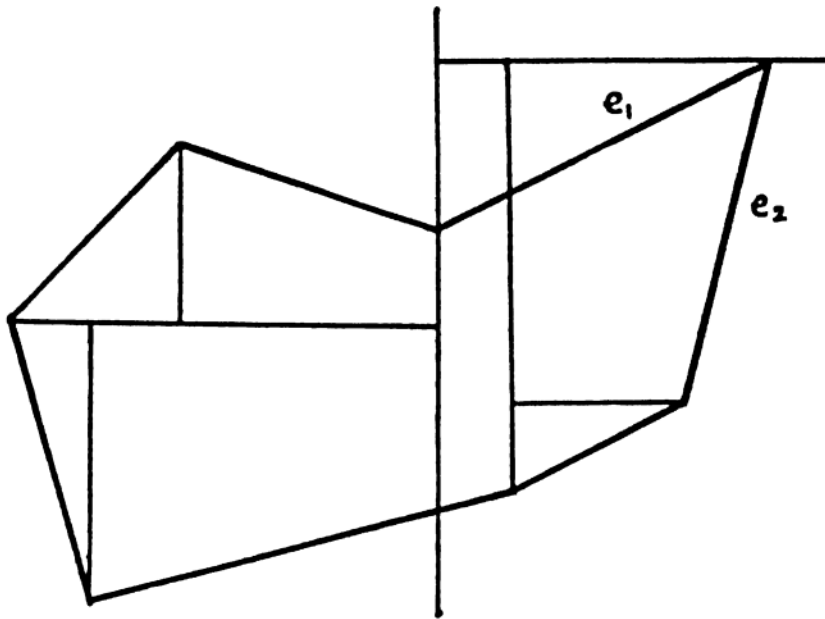


Fig. 3.5 : For this polygon, point membership cannot be answered through an edge k-d tree.

convex class and the monotone class.

The representation schemes that are used to store 2D object information, in some cases, can be extended to store 3D object information. So it is natural to ask whether the edge k-d tree concept can be extended to represent 3D polygons. The answer is in the negative. This is because, in the edge k-d trees, cuts parallel to one of the axes is made at vertices and the polygon is broken up into smaller and smaller polylines. But in the case of a 3D polygon, where should a cut be made? And, further, should the cut be made by a line or plane?

If we make the cut at a vertex, it may partition the faces in the middle and if the cut is made at an edge, the plane may not be parallel to any of the coordinate planes.

---

## CHAPTER 4

### Linear Octrees by Volume Intersection<sup>\*</sup>

---

The representation of 3D objects is essential for Computer Aided Design and Manufacturing (CAD/CAM), computer graphics, image understanding and solid modeling. But, unlike the representation of 2D objects, the representation of 3D objects is not that easy. This is partly due to the extra dimension involved and the lack of proper equipment to capture volumetric information from 3D objects. The extra dimension, depth, also causes a phenomenal rise in the space requirement for the storage of 3D objects. Further, capturing 2D, i.e., areal, information is relatively easy as a majority of the image acquisition systems come armed for capturing 2D data. Since the 2D image is acquired from a point outside the plane containing the 2D object, the entire image is captured easily. But unfortunately, such a thing cannot be done for 3D objects i.e., the 3D information about the 3D object cannot be captured from a point outside the space containing the object. Notwithstanding the inherent difficulty, the need for a 3D object representation has been realized and many methods have been devised to capture, store, retrieve and manipulate 3D object information. Some were presented in Sec.2.2. The most widely used 3D object representations are the octrees [Chen & Huang 88] and the Constructive Solid Geometry (CSG) [Voelcker & Requicha 77]. The octrees are preferred because of their conceptual clarity, ease of programming and the top-down

---

<sup>\*</sup> The main results have been published in CVCIP.  
cf. Lavakusha et al [89a]

paradigm. There are different techniques for building octrees: classified based on the format of their input data [Chen & Huang 88]. They are the spatial occupancy arrays, serial sections, silhouette images, depth images and object models. Of these, the silhouette images take the minimal amount of space, of  $O(N^2)$ . Obtaining the relevant volumetric information of a 3D object is not only difficult, but also involves processing large amounts of data. On the other hand, getting areal information is relatively easy as a majority of image acquisition systems are capable of acquiring 2D data. Hence it is generally accepted that using areal information to generate volumetric information would be cost effective than to directly acquire volumetric data. One such method is the volume intersection.

Hence, we have used this technique of building octrees from silhouette images, using the volume intersection.

#### 4.1 VOLUME INTERSECTION

The Volume Intersection (VI) is the technique of generating the 3D object representation from multiple silhouette images of the object. A silhouette image, or just, a silhouette of an object is defined as the binary 2D image obtained by the occluding contour of the 3D object uniformly filled with black pixels. For the 3D object that is to be represented, different silhouettes are obtained. Each of the 2D silhouette is then swept along the viewing direction to generate a 3D volume, called the swept volume. The different swept volumes obtained by sweeping the silhouettes are intersected to generate the 3D object representation. See Fig.4.1. For the volume intersection, the

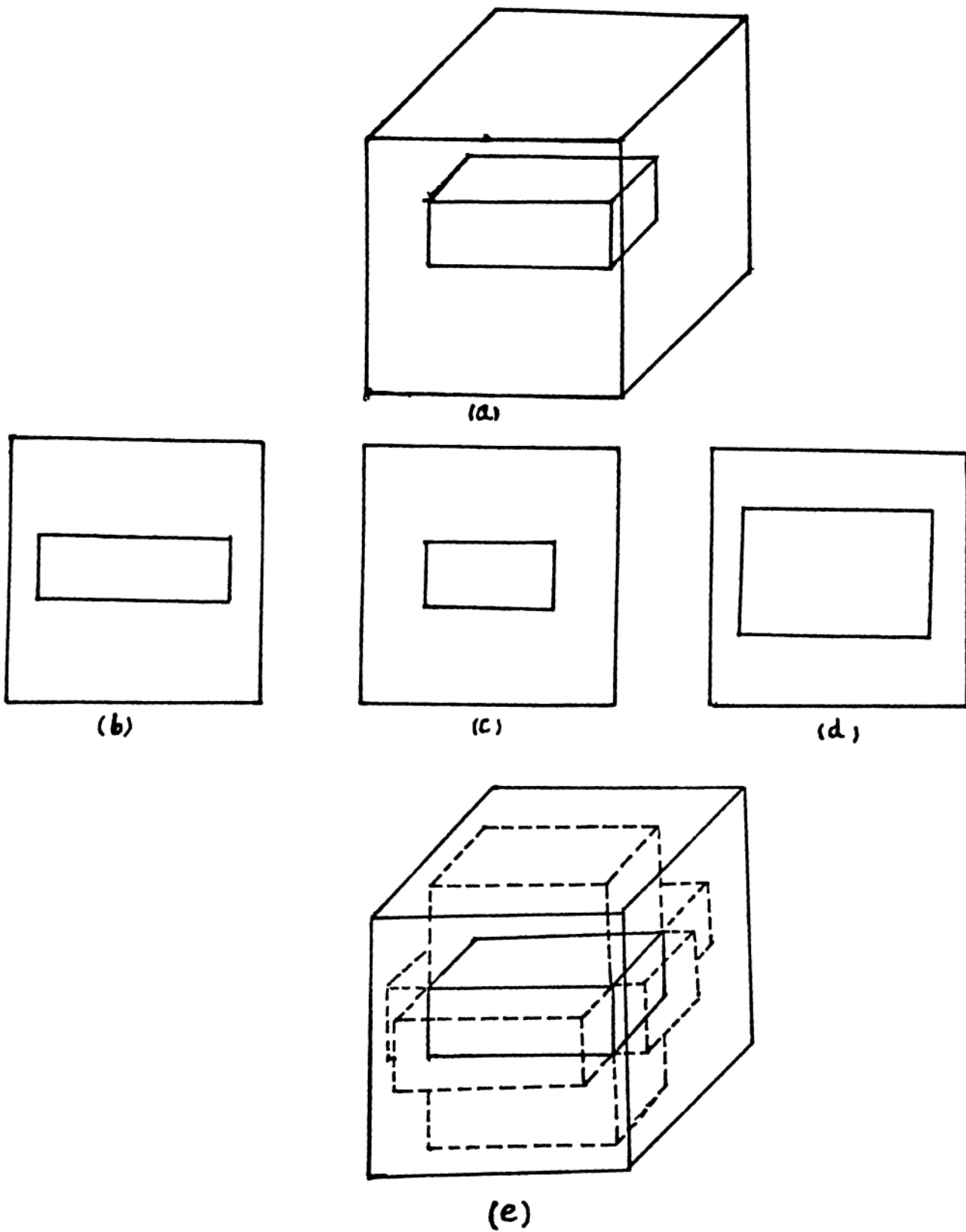


Fig. 4.1 : A 3D object (a), the front (b), side (c), top (d) silhouettes and the intersected volume of the cylinders (e).

silhouettes are easy to obtain and are stored as binary matrices of size  $N \times N$ . Generally, three mutually orthogonal silhouettes are taken viz the front, the side and the top. These are used to generate the swept volumes and then, are intersected to generate the 3D object representation. The volume intersection algorithm is so called because the swept volumes generated by sweeping different silhouettes of the 3D object along the respective viewing directions are intersected to build the object representation.

Here, it is assumed that the 3D object is enclosed in a cube of voxels, of size  $N \times N \times N$ ,  $N = 2^n$ , for some  $n$  ( $n$  is the resolution). Thus the volume intersection is to determine the  $N^3$  voxels based on its several projections of size  $N^2$ . However, the object reconstructed by the VI algorithm is only an approximation. The degree of approximation i.e., the accuracy of the reconstructed object, depends on the following factors:

(i) The properties of the 3D object : Consider a hollow sphere of radius 1, i.e.,  $x^2 + y^2 + z^2 = 1$  and the closed solid sphere of the same radius, i.e.,  $x^2 + y^2 + z^2 \leq 1$ . These two objects cannot be distinguished from one another from the outward appearances and as far as human vision is concerned these two are same. Hence, there is no meaning in reconstructing the hollow sphere by the VI technique. Specifically, an object with a cavity, which cannot be projected onto any silhouette cannot be reconstructed accurately. There are also certain type of objects, different from the above, which cannot be reconstructed exactly by the VI, even with a large number of views. That is mainly because certain portion of



the object cannot be projected. The object in Fig. 4.2 is well defined. However, the voxel with locational code 43, whether black or white, will always be realized as black by the VI algorithm, irrespective of the number of views and/or the viewing directions. There are some examples of 3D objects which cannot be accurately reconstructed by the VI method even if the number of views is very large. This is to show that the objects themselves play a crucial role in the accurate reconstruction.

(ii) The total number of views and the viewing directions :For the object in Fig 4.2, if the front, the side, and the top views are taken, the object reconstructed is the voxel cube of size 8. This is because all the three silhouettes are black squares of size 8. Veenstra and Ahuja [86] show how a better approximation is reconstructed by taking 13 silhouettes: 3 face-on, 6 edge-on and 4 corner-on views. An edge-on silhouette is obtained when the viewing direction is parallel to the plane of two axes of the octree space and bisects the angle formed by these two axes. Hence, an edge-on silhouette is specified by the two axes that

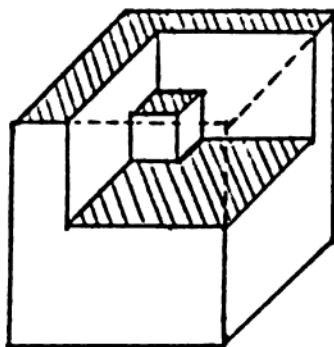


Fig. 4.2 : A 3D object that defies the volume intersection technique.

define it. A corner-on silhouette or an isometric silhouette [Yamaguchi et al 84] is obtained when the viewing direction is along the line joining a corner and the center of the initial cube and is defined by the corner through which the view is taken. Here, for the object in Fig. 4.2, the approximation obtained by considering the 13 views as proposed by Veenstra and Ahuja [86] is certainly a better one. The linear octree generated by this method is

00,04,02,06,05,07,48,50,54,56,53,57,28,68,38,78.

The approximation of the reconstructed object also depends on the direction of views. This can be done in two ways : keeping the object fixed and changing the directions of views, or fixing the directions and changing the orientation of the object. Of course, one is equivalent to the other. If the object cannot be displaced, the former is followed and if the cameras are fixed, the latter. For the object in Fig. 4.2, if the front silhouette is taken from an angle of  $-\pi/4$ , we get a better approximation, in which case the front silhouette will be as shown in Fig. 4.3.

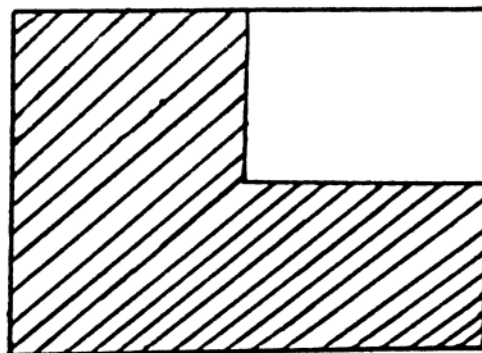


Fig. 4.3 : The silhouette taken from an angle of  $-45^\circ$  to the front of the 3D object in Fig. 4.2. It has an aspect ratio of  $\sqrt{2}$ .

Here, by just changing one direction of view, a better approximation is got. Also, observe that the convex hull of this silhouette is not a square, but a rectangle of  $8 \times 8\sqrt{2}$ . The aspect ratio is  $\sqrt{2}$  and this image is condensed to get a square of size 8. In this process of condensation, some floating point arithmetic is involved. Here, the benefit of doubt is given to black colour i.e., if a pixel is partially black, we make it black.

(iii) The resolution (pixel size) of the silhouettes : The resolution too has a role to play on the resulting 3D object. Let us consider  $S^2$ , the 2-sphere in  $E^3$ . The silhouette in any direction is a circular disc. Sweeping the silhouettes generate circular cylinders. The VI algorithm generates an object which is the intersection of a finite number of cylinders. If we take the sphere and the circular discs to be perfect, the VI paradigm will not give rise to a perfect sphere.

For example, take the three orthogonal silhouettes:  $x^2 + y^2 \leq r^2$ ,  $x^2 + z^2 \leq r^2$ , and  $y^2 + z^2 \leq r^2$  of the sphere  $x^2 + y^2 + z^2 \leq r^2$ . Clearly the point  $(r/\sqrt{2}, r/\sqrt{2}, r/\sqrt{2})$  is in each of the silhouettes but not in the sphere. But, by taking infinite number of silhouettes we can surely get the perfect sphere. However, in the domain of voxel based geometry, the sphere is never perfect since it is discretized by the voxels. Hence in this model of computation, we can only reconstruct an approximate sphere by a finite number of silhouettes and this number depends on the resolution factor. This phenomenon is not in the method, but it manifests itself in the underlying digitizing process, where one tries to discretize the continuous structures (smooth

curves/surfaces ). However great the resolution may be, still the digitization shows its effect on the pixel/voxel object. Although certain holes/cavities cannot be preserved in the reconstructed object representation, a better approximation can be generated by taking more than three silhouettes and/or a different set of silhouettes.

There are two methods of generating the swept volumes. The silhouette can be swept either orthographically or perspectively. In case of an orthographic projection, the swept volume is a cylinder and for perspective projection, a cone-shaped volume.

In this chapter, a top-down algorithm is proposed to build the linear octree from three mutually orthogonal silhouettes of an object. The orthographic projections are used and hence, usage of the words 'swept volume' and 'cylinder' interchangeably. In the next section, the linear octrees are examined in detail.

## 4.2 LINEAR OCTREES

Here the construction of the linear octree for a 3D object from the 3D spatial occupancy array is recalled, an voxel cube of side  $N$ ,  $N = 2^n$ . There are  $N^3$  voxels and each of the black voxel is encoded into an octal number, called the locational code [Gargantini 82a]. The locational code is a numeric index obtained from the coordinates of the voxel. The binary equivalents of the  $Z$ ,  $Y$  and  $X$  coordinates of the voxel are bit-wise interleaved and converted into a number in octal system. For example, the locational code of the voxel at  $(1,3,5)$  is 427. This is obtained by interleaving 101, 011 and 001, which yields 100010111 ie 427 in radix 8. After the generation of the locational codes for all

the black voxels, condensation [Samet 81b] is applied : if eight voxels' locational codes are same but for the last digit, then these eight are replaced by a code consisting of the common digits, followed by some kind of marker. Gargantini [82a] uses 'X' for this marker, but we use 8, which makes the locational code completely numerical rather than alphanumeric. For instance, if the locational codes 30 to 37 are all present, then they are substituted by 38. Further, if the locational codes 8,18,28,38,48,58,68 and 78 are all present, then they are replaced by 88. The list of locational codes of only black leaves is called the linear octree. A 3D object, its octree and linear octree are shown in Fig.4.4.

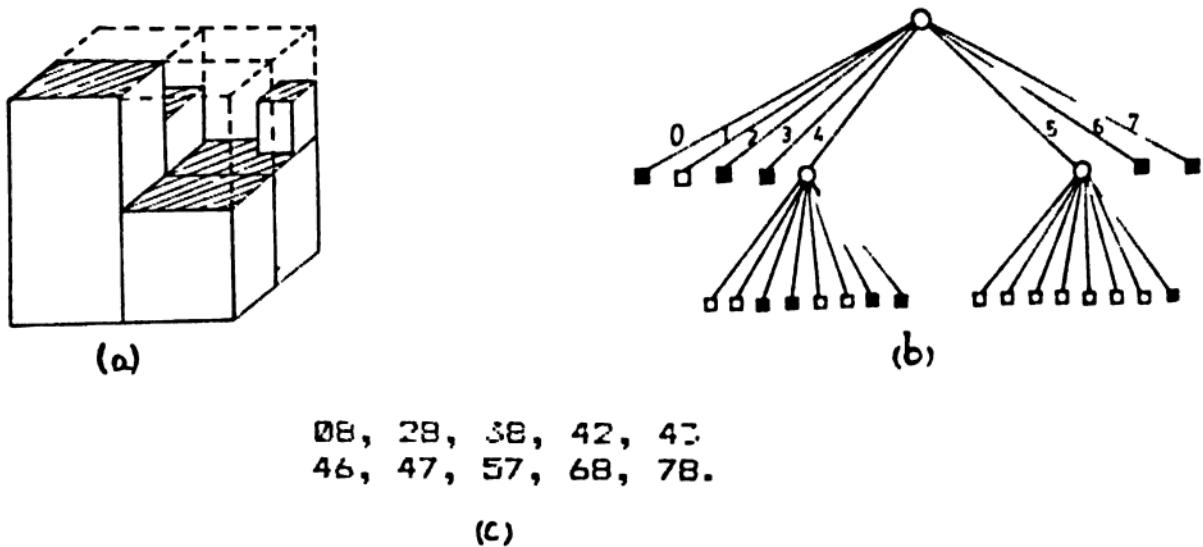


Fig. 4.4 : An object (a), its octree (b) and linear octree (c).

This is a bottom-up approach where the black voxels are encoded and then certain neighboring voxels are condensed and these octants in turn are condensed if they satisfy certain condition and so forth. i.e., we start from the voxel and by

enclosing neighboring voxels build up homogeneous octants. But, in the method proposed, the linear octree is generated using a top-down approach, where the locational codes of the largest permissible octants are computed first directly and then this part is logically removed from the black volume of the universe. And this process is repeated till the entire black volume is exhausted in a recursive manner. This economizes on two factors :

- (i) encoding of individual black voxels into locational codes
- (ii) condensation of voxels/octants into octants of bigger sizes.

Lemma 4.1 In the bottom up approach, to generate the locational code of an octant of size  $S$ ,  $S^3$  encodings, and  $\sum_{i=1}^{S-1} 8^i$  condensations are required (  $s = \log_2 S$  ).

Proof: In an octant of size  $S$ , there are  $S^3$  voxels and each of them are encoded and hence  $S^3$  encodings are necessary.

The number of octants of size 2 is  $8^{S-1}$  and hence  $8^{S-1}$  condensation of  $S^3$  voxels is required. In general, the number of octants of size  $2^i$  is  $8^{S-i}$ ,  $0 \leq i \leq s$ . Since from the voxels, we generate octants of size 2 first, and then octants of size 4 and so on, the number of condensations is

$$8^{S-1} + 8^{S-2} + \dots + 8 \quad \text{i.e.,} \quad \sum_{i=1}^{S-1} 8^i \quad \text{QED}$$

Abel and Smith [83] and Samet [85b] have proposed alternate methods of generating locational codes in the context of storage for 2D objects i.e., linear quadtrees. These can easily be extended to linear octrees. The linear quadtrees have been detailed in "Pointerless quadrees" ( cf. 2.1.2.5.2). The

pointerless means of storing octrees came into being in '82, with the sensational paper by Gargantini [82a].

Some of the merits of linear octrees over pointered octrees are [Gargantini 82a]:

- pointers are eliminated, thereby saving memory space as well as access/computational time.
- space and time complexities depend on the number of black nodes only, because only black nodes are stored.
- the locational code gives the complete information of the node it represents viz the location, the size, the locational codes of its neighbors, relation between two codes etc., and
- dynamic in nature.

#### 4.3 LINEAR OCTREES THROUGH VOLUME INTERSECTION

In this section the new top-down method of generating the linear octree through the technique of volume intersection from three mutually orthogonal 2D silhouettes of the 3D object is presented. The octree space is a cube of size  $N$ ,  $N = 2^n$ , for some  $n$ . Hence, the locational code of any black voxel has  $n$  digits and the 'don't-care' digit is denoted by 8. The input to the algorithm is the three 2D binary square arrays of size  $N$  : `FRONT`, `SIDE` and `TOP` representing the respective silhouettes of the 3D object. The top left corner pixel in the 2D binary arrays is at  $(0,0)$  and the bottom right corner, at  $(N-1,N-1)$ . Further, a quadrant is defined by its zero-corner coordinates (i.e., its top left corner coordinates) and its size. The output is the list of locational codes of all black octants constructed by a top-down

method that make up the 3D object.

#### 4.3.1 THE ALGORITHM

The FRONT is used as the reference silhouette and the other two, the SIDE and the TOP as the supporting silhouettes. From the two supporting silhouettes, the runlengths of black voxels are generated for each of the black pixels in the reference silhouette, which is stored in the array `range` of size  $N \times N \times 2$ . For each column  $j$  of TOP,  $1 \leq j \leq N$ , the two integers  $k_1$  and  $k_2$  are calculated such that the  $(k,j)$ th pixel of TOP is black for  $k_2 \leq k \leq k_1$ . And for each row  $i$  of SIDE, the two integers  $l_2$  and  $l_1$  are calculated such that the  $(i,k)$ th pixel of SIDE is black for  $l_1 \leq k \leq l_2$ . Thus the range of black voxels for the black pixel at  $(i,j)$  of FRONT is a pair of integers

$$\text{range}[i,j,0] = \max(l_1, k_1)$$

$$\text{range}[i,j,1] = \min(l_2, k_2).$$

This concept is illustrated in Fig.4.5.

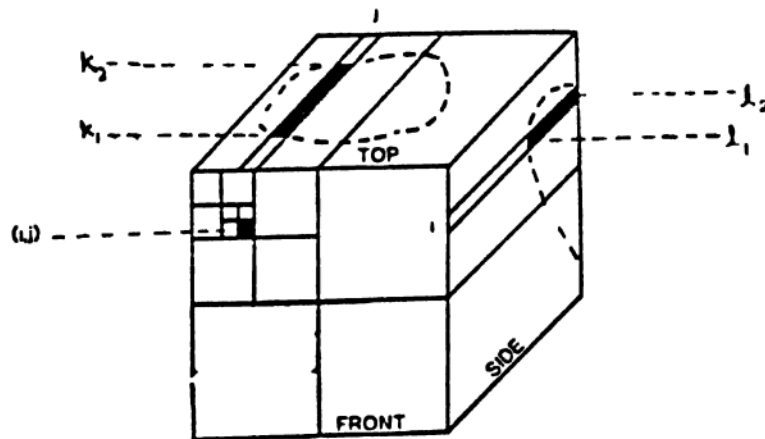


Fig. 4.5 : 'range' calculation of  $(i,j)^{\text{th}}$  element of FRONT.



Then the `FRONT` is scanned in a Warnock-type fashion that uses successive subdivision of the image until each quadrant is homogeneous. Warnock's visible surface algorithm is discussed by Newman and Sproull [84] (in the context of hidden surface elimination). For each black quadrant of `FRONT` of size  $S$ , so generated, the black portion of the volume is obtained by sweeping the quadrant using the runlength information. From this black portion, we keep generating the locational codes of the largest octants permissible. After this, if necessary, the quadrant is bisected along the  $X$  and  $Y$  axes to obtain four quadrants of size  $S/2$ . This process is repeated till the voxel level is reached or the black portion is completely scanned.

This is the top-down approach wherein octants of the largest size permissible are constructed first and these are removed from the black portion of the universe. The same process is applied to the remaining part of the black portion.

Our method makes use of the functions `SCANMAT`, `LOOKAT`, `CONSTRUCT` and `ALL_CONSTRUCT`. The function `SCANMAT` scans the `FRONT` image in a Warnock type method to generate the black quadrants. For each quadrant of size `size`, with upper left corner at  $(X,Y)$ , of the `FRONT` image, `LOOKAT(X,Y size)` returns 0, 1 or 2 depending on whether the quadrant is white, black, or gray, respectively. `CONSTRUCT` generates the locational codes of octants of size  $S$ , if any, and then invokes itself again four times, if necessary, to generate locational codes of octants of size  $S/2$  in the remaining part of the black portion of the swept volume. The function `ALL_CONSTRUCT` is invoked by `CONSTRUCT` for generating locational codes of octants of size  $S/2$ . A detailed algorithm in

C-like pseudo-code, is given in Alg. 4.1.- Alg. 4.4.

```
-----
main ( )
begin
/* length of black voxels corresponding to the black pixel at
   (i,j) of FRONT */
for i = 0 to N-1 step 1 do
    determine depthrange[i,0] and depthrange[i,1] from runlength
        in TOP;
for row = 0 to N-1 step 1 do
    compute l1 and l2 from SIDE;
    for col= 0 to N-1 step 1 do
        if (FRONT[row,col] is black) then
            begin
                range[row,col,0] = max(l1,depthrange[col,1]);
                range[row,col,1] = min(l2,depthrange[col,0]);
            end;
SCANMAT (N,0,N);
end.
-----
```

#### Algorithm 4.1

The function SCANMAT (X,Y,SIZE) scans the quadrant at (X,Y) of size SIZE of the FRONT image. If it is black, invokes CONSTRUCT( ) to generate the locational codes.

```
-----
SCANMAT (X,Y,SIZE)
begin
if (SIZE > 0) then
    begin
        l = LOOKAT(X,Y,SIZE);
        if (l = 1) then                /* the quadrant is black */
            CONSTRUCT(X,Y,range[X-SIZE,Y,0],range[X-SIZE,Y,1],SIZE,3);
        elseif (l = 2) then            /* the quadrant is gray */
            begin
                sb2=SIZE/2;
                SCANMAT (X,Y,sb2);
                SCANMAT (X+sb2,sb2);
                SCANMAT (X,Y+sb2,sb2);
                SCANMAT (X+sb2,Y+sb2,sb2)
            end ;
    end ;
end.
-----
```

#### Algorithm 4.2

In the function CONSTRUCT(X,Y,A,B,S,flag), the flag determines the subrange of (A,B) that is to be searched for octants of size S.

```

-----
CONSTRUCT (X,Y,A,B,S,flag)
begin
  if (A ≥ B) return;
  if (S = 1) /* at voxel level */
  begin
    case (flag) of
      begin
        1 :A = range (X,Y,0)
        2 :B = range (X,Y,1)
        3 :A = range (X,Y,0)
          B = range (X,Y,1)
      end;
    for i = A to B step 1 do
      print the locational code of the voxel at (X,Y,i);
  else /* i.e., s ≥ 1 */
    compute m0max,m0min,m1max,m1min;
    /* where m0max = max range [Xi,Yi,0]
      1 ≤ i ≤ 4
      m0min = min range [Xi,Yi,0]
      1 ≤ i ≤ 4
      m1max = max range [Xi,Yi,1]
      1 ≤ i ≤ 4
      m1min = min range [Xi,Yi,1]
      1 ≤ i ≤ 4
      (Xi,Yi), 1 ≤ i ≤ 4 are the four corner pixels of the quadrant */
    determine t0 and t1;
    /* the least and greatest multiples of S in the
      interval ( m0max,m1min) */
    if (t0 ≥ t1) then /* there are no octants of size S */
      ALL_CONSTRUCT (X,Y,A,B,S/2,flag);
    else
      begin
        limit = log2 S; /* the number of 8's in the
          locational code for this octant */
        for i = t0 to t1 step S do
          print the locational code of the voxel at
            [X,Y,i] after replacing the least significant
            'limit' number of digits by 8's ;
          if (t0 ≠ m0max or t0 ≠ m0min) then
            ALL_CONSTRUCT (X,Y,A,t0,S/2,1);
          if (t1 ≠ m1min or t1 ≠ m1max) then
            ALL_CONSTRUCT (X,Y,t1,B,S/2,2);
        end;
      end;
  end;
end.
-----

```

### Algorithm. 4.3

If the function CONSTRUCT(X,Y,A,B,S,flag) finds that there are no octants of size S, the quadrant at (X,Y) is quadsected and CONSTRUCT( ) is invoked four times, one for each of the four

quadrants, to generate location codes for octants of size  $S/2$ . This is performed by the function ALL\_CONSTRUCT.

```

-----
ALL_CONSTRUCT (X,Y,A,B,S,flag)
begin
    CONSTRUCT (X,Y,A,B,S,flag);
    CONSTRUCT (X+S,Y,A,B,S,flag);
    CONSTRUCT (X,Y+S,A,B,S,flag);
    CONSTRUCT (X+S,Y+S,A,B,S,flag);
end.
-----

```

#### Algorithm 4.4

An interesting pattern is observed in the locational codes of the voxels in the swept volume generated by any pixel of FRONT image. The locational code of the voxel at  $(X,Y,i)$  is the sum of the locational code of the voxel at  $(X,Y,0)$  and the `offset[i]`, where

$$\text{offset}[i] = 4 * (\text{binary equivalent of } i)$$

Since `offset[ ]` is used very frequently, these numeric indices are precomputed and stored in an array. The first few elements of the array 'offset' are 0,4,40,44,400,404, etc. See Fig.4.6. This comes in quite handy when we have to generate the locational codes of many voxels in a runlength. In addition to that, the locational code of an octant of size  $S$ , will have  $\log_2 S$  number of 8's. So, to compute the locational code of an octant of size  $S$ , instead of generating the locational codes of all the  $S^3$  voxels and then condensing them (as would be done in a bottom-up approach), here we compute the locational code of the voxel at the zero-corner of the octant (or any other voxel inside the octant would be as good) and replace the  $\log_2 S$  least significant digits by 8's. This top-down approach saves a significant amount of time.

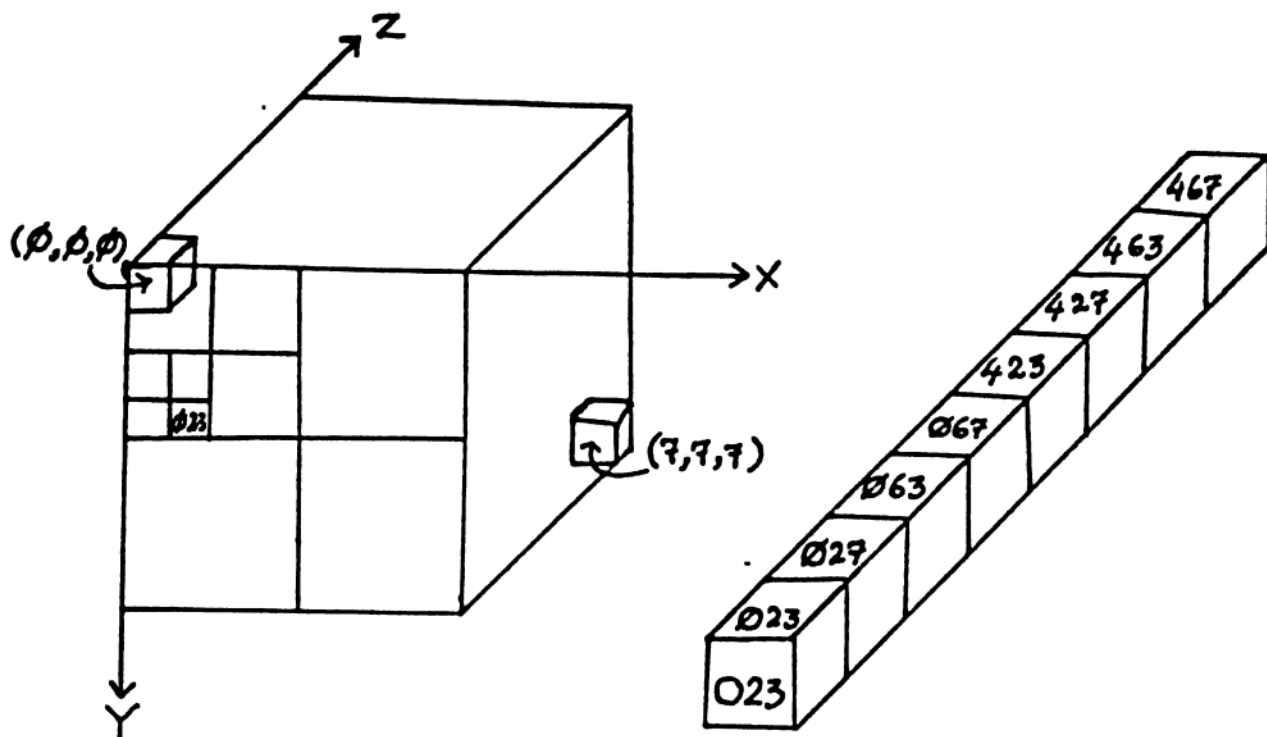


Fig. 4.6 : The runlengths of a voxel and their locational codes.

#### 4.3.2 ANALYSIS

The space and time complexities of our algorithm are presented here.

**Theorem 4.1** The algorithm takes  $O(N^2)$  space and  $O(N^2 + R)$  time to construct the linear octree with  $R$  nodes.

**Proof:** The only data structure that has been used is the array. We have used four arrays: two  $N \times N$  arrays for storing the FRONT and the SIDE. The TOP silhouette is read and the array depthrange of size  $N \times 2$  is computed and into the same array the SIDE silhouette is stored. The array range, is of size  $N \times N \times 2$ , that stores the two extreme coordinates of runlengths of black voxels corresponding to the black pixels of the reference silhouette. The offset is an array of size  $n$ . This amounts to a space

complexity of  $O(N^2)$ .

The algorithm makes use of the functions MAIN, SCANMAT, LOOKAT, CONSTRUCT and ALL\_CONSTRUCT. The function MAIN, generates the range and hence, has a time complexity  $O(N^2)$ . The function SCANMAT scans the FRONT matrix and if the quadrant is black, invokes the function CONSTRUCT and if gray, invokes SCANMAT four times in a recursive manner. If there would be T nodes in the quadtree of the FRONT, SCANMAT would be called T times. And for each black node in the FRONT quadtree CONSTRUCT is invoked once. CONSTRUCT generates the locational codes of all the black octants of the maximum size from the swept volume corresponding to that quadrant, and ALL\_CONSTRUCT is invoked if any black portion remains in the swept volume. Hence if there are R nodes in the linear octree, CONSTRUCT is called at most R times and hence the complexity is  $O(R)$ . Thus the total time requirement for functions SCANMAT, LOOKAT, CONSTRUCT and ALL\_CONSTRUCT is  $O(R)$ . Together with MAIN, the total complexity is of  $O(N^2 + R)$ . QED

#### 4.4 OCTREES THROUGH VOLUME INTERSECTION

The octree, an extension of quadtree to the third dimension and a tree of outdegree eight, is a hierarchical data structure for a compact representation of 3D objects. More details are given in Sec 2.2.1.3. Here an existing method [Chien and Aggarwal 86a] of generating the octree of a 3D object from three mutually orthogonal silhouettes through volume intersection is recalled. From each of the three discrete binary 2D images, a pointered quadtree is obtained. The volume obtained by orthographically sweeping the 2D silhouette in the

corresponding viewing direction is represented by a pseudo-octree. The pseudo-octree, computed from the quadtree in a straightforward way, is a compact version of the corresponding octree, making use of the fact that the swept volume is symmetrical with respect to a plane normal to the viewing direction. The pseudo-octree is, in fact, a quadtree in which every node denotes two octants. Every child-node of a gray node in this quadtree is assigned two numbers that are the numbers of the corresponding identical child-nodes in the associated octree. The correspondence between the labeling of a quadtree and the numbering of the corresponding pseudo-octree is shown in Table 4.1

| VIEW  | NW  | NE  | SW  | SE  |
|-------|-----|-----|-----|-----|
| Front | 0,4 | 1,5 | 2,6 | 3,7 |
| Top   | 4,6 | 5,7 | 0,2 | 1,3 |
| Side  | 0,1 | 4,5 | 2,3 | 5,7 |

Table 4.1 Correspondence between labeling of quadtree and numbering of the corresponding pseudo-octree

The volume intersection technique is applied to the three pseudo-octrees as follows : the three trees are traversed in parallel and the nodes of the resulting octree are generated. If any of the nodes in the corresponding location of the three trees is white, then the resulting node is white. If all the three are black, the resulting node is black. Else it means that at least one of the nodes is gray. If a node is gray and other two, black, that gray node of the pseudo-octree is converted to an octree

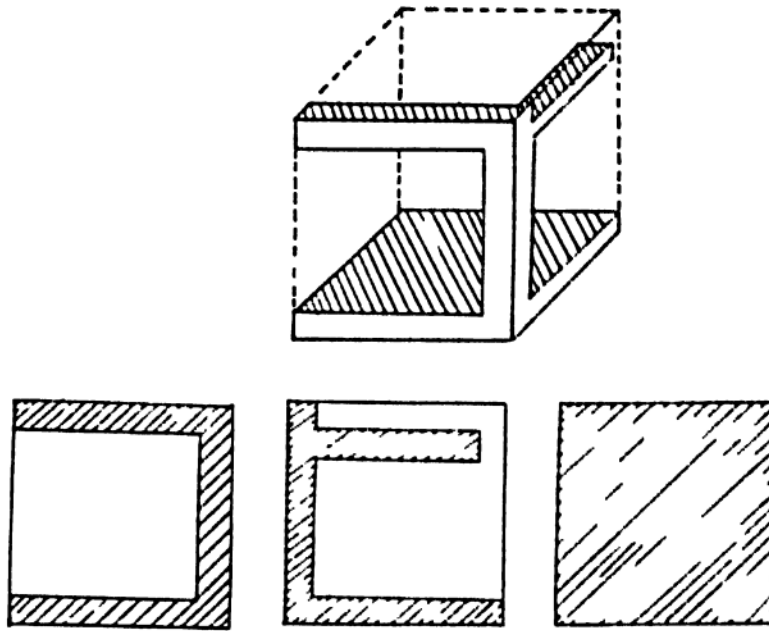


Fig. 4.7 : An object and its three silhouettes to exhibit the necessity of the condensation of white nodes in the octree. The node, '4' is to be condensed.

node. If two are gray and one is black, the two gray nodes are intersected. If all the three are gray, the function is applied again, in a recursive manner.

In this process, the resulting octree so generated can have a white node partitioned as well. This is because each of the recursive calls to the function that intersects the three nodes may result in a homogeneous node and all the eight of them could be white. In such situations these child-nodes must be condensed to get a proper octree. In Fig. 4.7, an example that exhibits the necessity of condensation is shown. Thus, there are three major steps involved in this process, namely

- generation of quadtrees and pseudo-octrees
- intersection of swept volumes
- condensation of the resulting representation to obtain an octree.



But our algorithm for volume intersection, directly generates the linear octree and hence condensation is not required. Moreover, the runlengths of black voxels corresponding to each black pixel in the FRONT are computed in the beginning itself and hence operations are done only on the black portion. This saves a considerable amount of time because no processing is done on the white portion of the universe. In Chien and Aggarwal's [86a] method of constructing the octree, runlengths of black voxels could be spread over a wide branch of the tree at at the lowest level and many pointer references have to be made to access them.

Both the algorithms, Chien and Aggarwal [86a] and ours were coded in C language and were run on a OMC-58000 mini computer system in a UNIX -like environment. The images were arbitrarily chosen, such as a nut, tea cup, torus etc. The input to both the programs were the FRONT ,the SIDE and the TOP silhouettes of size 64 x 64. The computational times are reported in Table 4.2.

| 3D-OBJECT    | Processing time for generation of |                     |
|--------------|-----------------------------------|---------------------|
|              | Octree (min:sec)                  | Linear Octree (sec) |
| Nut          | 6.7                               | 2.8                 |
| Tea cup      | 1:02.7                            | 4.9                 |
| Torus        | 1:03.9                            | 3.5                 |
| Goblet       | 2:08.4                            | 4.2                 |
| Paper weight | 8:28.4                            | 6.2                 |

Table 4.2 Computational time taken to generate the octree and the linear octree

## 4.5 CONCLUSIONS

In this chapter we have presented an algorithm to reconstruct the 3D object from multiple 2D silhouettes of the 3D object using the VI technique in a top-down manner. More specifically, we considered three mutually orthogonal silhouettes of the object, generated the swept volumes by sweeping the silhouettes orthographically along the respective viewing directions, and then intersected these volumes. We have used the `FRONT` as the reference silhouette and the other two as the supporting silhouettes. For each black pixel in the `FRONT`, we generate the runlength of black voxels. Thus, we isolate the black portion and perform all operations only on this black portion. For each black quadrant in the `FRONT`, the maximum possible octants' locational codes are generated and then do the same for the remaining black portion, if any, in the swept volume generated by this black quadrant, in a top-down recursive manner.

Gargantini [82a] generates the linear octree by first generating the locational codes of all black voxels and then condensing them. This is a bottom-up approach. The top-down method that we proposed, not only avoids the encoding process for each voxel but also does away with the condensation of voxels'/octants' locational codes to octants of bigger size(s).

The construction of a linear octree using this method is shown to be superior to that of an octree [Chien and Aggarwal 86a]. Both the algorithms were coded in C language and the running times are reported in Table 4.2. The timings speak for themselves. However, the algorithm to generate the octree [Chien and Aggarwal 86a] was modified. We did not generate the pseudo-

octrees from the quadtrees of the silhouettes. We got the octree without the phase of pseudo-octree generation, but had to sacrifice the length of the source code. It became lengthy because each of the combination had to be specified separately. The time complexity for reconstructing the octree, as proposed by Chien and Aggarwal [86a] is of  $O(R^3 \log R)$ , where  $R$  is the number of nodes in the octree, but for our method it is only of  $O(N^2 + R)$ .

For our algorithm to work, we constrain that the object be convex. This algorithm, as such, works for a broader class of objects. The constraints are not on the object but on the three silhouettes. The `FRONT` silhouette can be anything, but the `SIDE` and the `TOP` silhouettes should be monotonic along the line passing through the center of the silhouette and parallel to the  $Z$ -axis. This ensures that the runlengths of black pixels in the `SIDE` and the `TOP` silhouettes are single intervals. The algorithm can be modified to suit the case where the runlengths are not single intervals of black pixels, but a disjoint union of intervals, in which case the array `range` will be a linked list of intervals and the program can be modified accordingly. Due to the use of limited information as input, the representation of the 3D object obtained by the process is not the exact one, but only an approximation. One of the drawbacks of the VI paradigm is that the excess volumes corresponding to the cavities of the object will never be removed from the reconstructed object, if the cavities cannot be projected into the silhouettes [Potemsil 85]. But one would like to know how good the approximation is and if there is a measure to determine the proximity of the

approximation to the original object. Over and above that, one would also ask if there is a method to improve the approximation automatically i.e., can some new direction of view be determined to enhance the approximation. These aspects are discussed in the next chapter.

## Volume Intersection Algorithm with Changing Direction of Views \*

The object reconstructed by volume intersection from three mutually orthogonal silhouettes is only an approximation. In Chapter 4, how the approximation depends on the object is shown; the number and/or directions of views and the pixel resolution. One of the simplest and the most obvious schemes for determining the viewing directions is to take three mutually orthogonal views, namely the front, the top and the side views. In the previous chapter, an algorithm was presented to generate the linear octree of a 3D object from the binary arrays of 2D silhouettes of these three views. But in general, three views are usually not enough to obtain a good reconstruction of the 3D structure of an object. On the other hand, the volume intersection algorithm with any set of arbitrary viewing directions will not be as simple and efficient as is the case with the three views. Veenstra and Ahuja [86] show that like these three views, there are other views such as edge-on and corner-on views. The correspondence between the quadrants of the silhouettes and the octants of the object, when the silhouette is taken from one of the face-on, edge-on or corner-on views is well defined. For the front silhouette, the octants 0 and 4 are projected onto quadrant 0, 1 and 5 onto 1, 2 and 6 onto 2, and 3 and 7 onto 3. The mapping for the three face-on views is given in Fig. 5.1, where the labels of the quadrants and the

---

\* The main contents of this Chapter have been presented in an International Conference. See [Lavakusha et al 89b].

corresponding labels of the octants, in brackets, are given.

|            |            |
|------------|------------|
| 0<br>(0,4) | 1<br>(1,5) |
| 2<br>(2,6) | 3<br>(3,7) |

|            |            |
|------------|------------|
| 0<br>(1,0) | 1<br>(5,4) |
| 2<br>(3,2) | 3<br>(7,6) |

|            |            |
|------------|------------|
| 0<br>(4,6) | 1<br>(5,7) |
| 2<br>(0,2) | 3<br>(1,3) |

Fig. 5.1 : The correspondence between labels of the quadrants and the octants.

An edge-on view is one, when the viewing direction is parallel to the plane of two axes of the octree space and bisects the angle formed by these two axes. Here an edge-on view is specified by the two octants through which the direction of view passes. Fig. 5.2 shows the 1-3 view, which is a rectangular region of aspect ratio  $\sqrt{2}$ . Since opposite edge-on viewing directions result in the same silhouette, there are only six edge-on views. In the edge-on view, some octants obscure other

|   |   |   |
|---|---|---|
| 0 | 1 | 5 |
| 2 | 3 | 7 |

Fig. 5.2 : The 1-3 edge-on view.

octants lying behind and result in overlapping silhouettes. For example, in Fig. 5.3, the silhouette image of octant 1 partially overlaps the octants 0 and 5 and completely overlaps the octant 4. In view of this, there are 2 completely hidden ( octants 4 and 6 ), 4 partially hidden ( 0,2,5,7) and two completely visible ( 1 and 3 ) octants. As a result of this partial occlusion, the

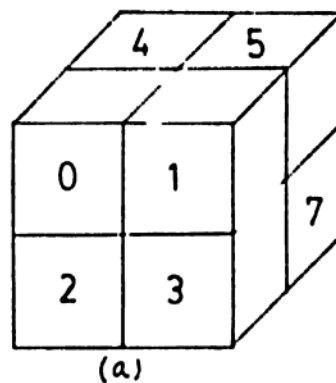


Fig. 5.3 : The hidden (4,6), partially hidden (0,2,5,7), and visible (1,3) octants.

octree refinement process for an edge-on view is more complicated than that of a face-on view. Veenstra and Ahuja [86] proposed a two phase approach. In the first phase, the silhouette of a parent octant is partitioned into four equal rectangular regions, which represent the four partially hidden octants. See Fig. 5.4a. Then in the second phase the silhouette is decomposed again into six different regions so that the two middle regions represent the images of the remaining four octants. See Fig. 5.4b. Hence in the octree refinement process when an edge-on view is used, the color of the nodes in the octree is updated in two separate phases.

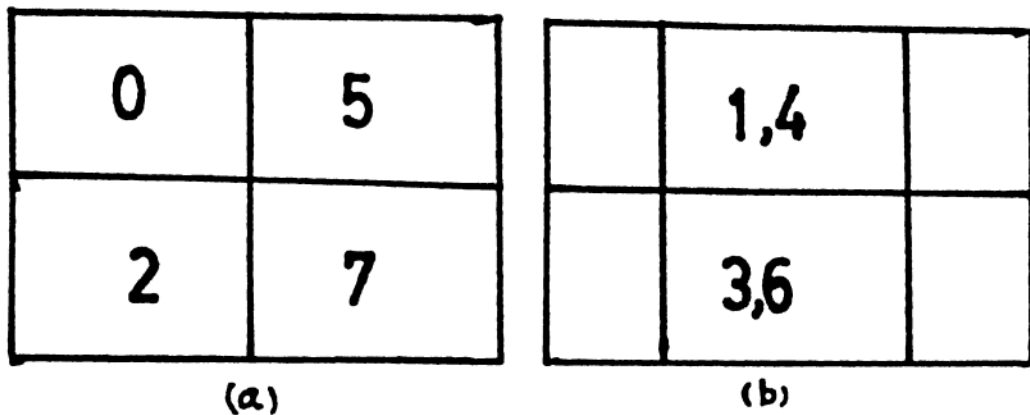


Fig. 5.4 : Partitioning of the 1-3 edge-on view, (a) for partially hidden octants and (b) for the other four octants.

A corner-on view (Yamaguchi et al [84] call it isometric view ) is a view when the direction of the view passes through a corner and the center of an octant. A corner-on view is specified by the corner that defines it. The corner-on view of a cube is a regular hexagon. Altogether, there are four different corner-on views ( since silhouettes from diagonally opposite corners are same ). It is to be observed that the projections of the eight octants partition the hexagon into 24 equilateral triangles and that each octant projects onto six such triangles. Because of the overlapping, a triangle in the hexagon can correspond to one, two or four octants. The correspondence for the corner-1 view is shown in Fig. 5.5, where numerals in the triangle represent the octants that project onto that triangle. To process the corner-on view Veenstra and Ahuja [86] proposed to group the 24 triangles into six larger triangles and represent each by a triangular quadtree. See Fig. 5.6. In a triangular quadtree the leaf nodes are equilateral triangles rather than squares. The refinement of the octree is then



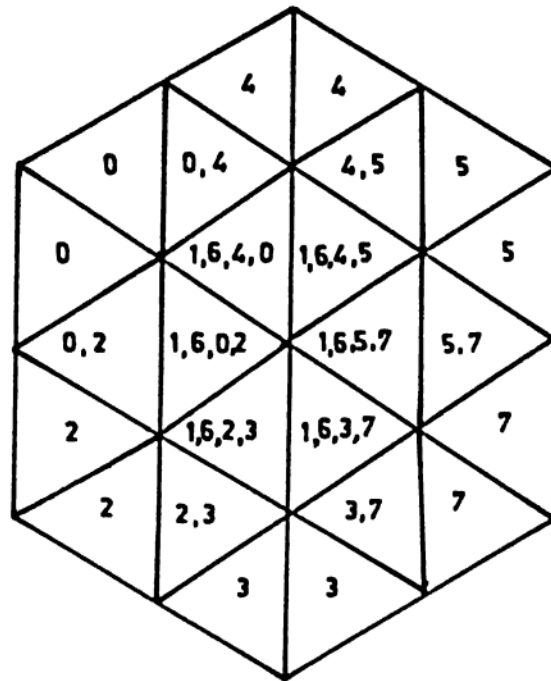


Fig. 5.5 : The silhouette of the 1-corner view.

performed by visiting the corresponding nodes in the triangular quadtrees for each octree node.

The refinement of an octree is simple when the correspondence between the octants of the object and the quadrants of the silhouettes is available. Since this is done by

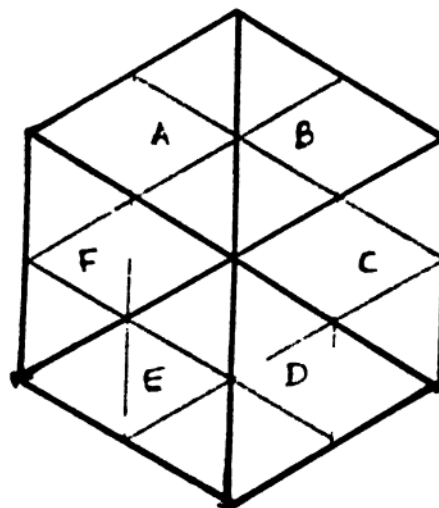


Fig. 5.6 : Division of the 1-corner view into six triangles.

traversing the octree and checking the colors of the quadrant corresponding to the octant visited, no intricate geometrical intersection is involved. Only simple Boolean intersections are performed at each step. Hence, the octree can be computed without intricate projection or, geometric operations. Thus, when the 3D object is represented by an octree and the silhouettes, by quadrees, the volume intersection technique can be carried out efficiently for the three face-on, the six edge-on and the four corner-on views providing a more accurate reconstruction.

However, for certain kind of objects, the set of these thirteen views may not provide a strict improvement over the set of three views. For the object in Fig. 5.7, three views are necessary and sufficient. The object is totally reconstructed from the front, side and top silhouettes.

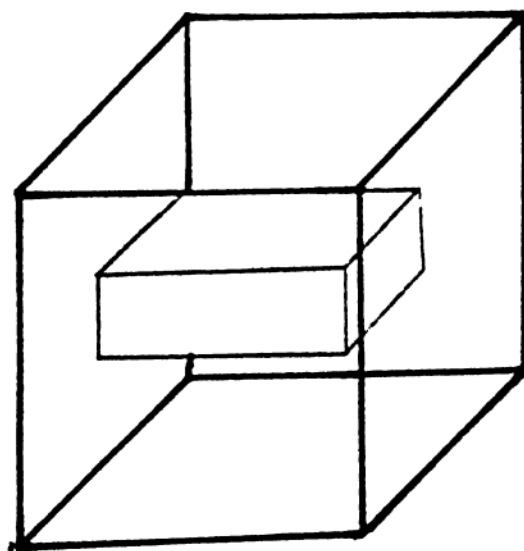


Fig. 5.7 : An object totally reconstructible from the three face-on views.

A viewing direction is said to be redundant if the silhouette obtained from that viewing direction does not improve

the approximation of the reconstructed object. For the object shown in Fig. 5.8, the three face-on views, the edge-on view 1-3 and the corner-3 view are sufficient to completely rebuild the object using the volume intersection method. Any other viewing direction will only be redundant. In such a case it is unnecessary to carry out extra computation for the redundant viewing directions. Thus if, at any stage it is known *a priori* that no more improvement can be attained by any additional views then the algorithm may be terminated, instead of considering all the thirteen views always. Thus considerable computational efforts can be saved.

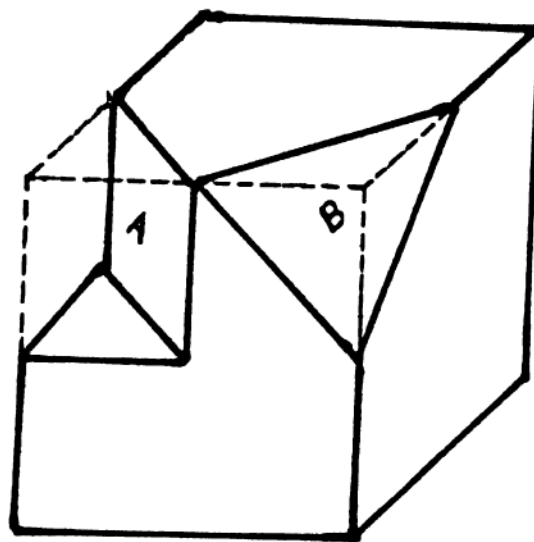


Fig. 5.8 : An object for which the three face-on views are not sufficient and the 13 views are more than necessary. The face 'A' is at an angle of  $45^\circ$  to the front and the face 'B', at  $45^\circ$  to the plane  $xz$ .

In another type of situation, it is also possible that a set of a smaller number of different viewing directions may result in a better reconstruction. For the object illustrated in Fig. 5.9, the three silhouettes taken from directions normal to the three intersecting faces will completely reconstruct the object,

whereas the predefined set of the 13 views will not. For any arbitrary viewing direction, other than the 13 views mentioned above, the correspondence between the quadrants of the silhouette and the octants of the object is not well defined and hence intricate geometric intersections and floating point operations are necessitated. Although the algorithm turns out to be more complicated for any arbitrary directions of view, it may still be cost-effective if a single such direction replaces a number of other prespecified directions.

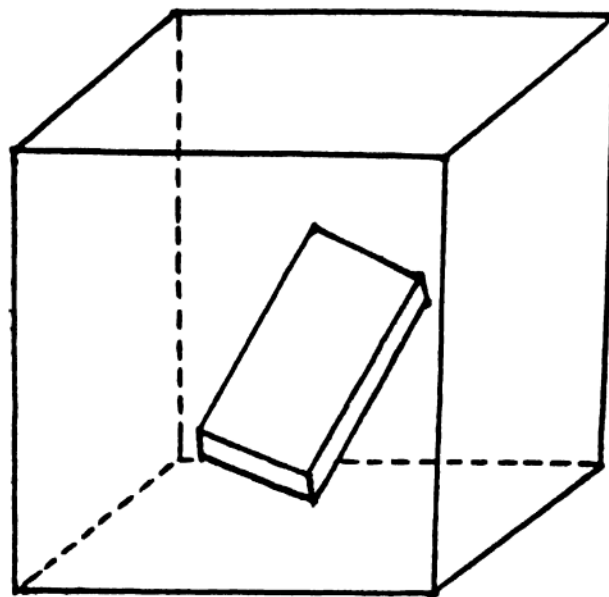


Fig.5 9

Motivated by these observations, we address the problem of automatically selecting some additional viewing directions to get a good reconstruction of the 3D object. To the best of our knowledge, there is no previous effort to introduce such a learning concept in vision system programming related to volume intersection. These problems are formalized below.

## 5.1 PROBLEM SPECIFICATION

Given a set,  $P$ , of viewing directions, the volume intersection algorithm generates an approximate reconstruction of the original object. Let us call this as the VI-hull with respect to  $P$  (it is some sort of visibility hull with respect to  $P$ ). The VI-hull may be the object itself or, may properly contain the object. Here the following problems are addressed.

- a : For a given set  $P$ , is the VI-hull, the object itself?
- b : If the VI-hull is different from the object, then can we append  $P$  with additional views so that the updated VI-hull provides a more accurate reconstruction? How do we update the VI-hull with any arbitrary direction  $a$  ? How do we determine this additional direction of view ?
- c : How do we quantify the degree of accuracy of the object reconstructed ?

It is to be noted here that the degree of approximation in the scheme of volume intersection is a function of many parameters, the most important being (a) the geometrical features of the object and (b) the set of viewing directions. For certain objects with holes and/or cavities it is not possible to reconstruct it beyond a certain degree of accuracy even with arbitrarily large number of views. This is due to the geometrical features of the object.

We are here concerned with determining the additional viewing directions assuming the difficulty inherent due to the geometrical properties of the object.

In this chapter we attempt to solve the problems a - c. As

in Chapter 4, it is assumed that the silhouettes of different viewing directions are represented by binary arrays and the 3D object is reconstructed as a linear octree. Beginning with a set  $P$  of three silhouettes the algorithm described in the previous chapter is applied to get an initial estimate of the 3D object. An algorithm is proposed here to refine this reconstruction with additional silhouettes. To make the presentation easy only directions of view, that lie in one of the three coordinate planes are considered. A method is proposed to determine the new directions of view and finally define a measure to quantify the accuracy of the reconstruction.

## 5.2 THE ALGORITHM

In this section we describe the important modules of the proposed algorithm. The initial estimate of the object structure can be obtained by taking  $P$  having three orthogonal viewing directions namely the front, the top and the side. The initial estimate of the object is derived in the form of *range*, from the front, side and top silhouettes, as described in Chapter 4. A method is presented to update the VI-hull (represented in the form of *range*) with additional silhouettes. In the current VI-hull, the voxels which do not have any ambiguity are identified in the next part of the algorithm and using this concept, a method is proposed to determine a new additional direction and also to measure the degree of accuracy of the reconstruction. Finally, a linear octree is generated as the final reconstruction from the intermediate structure *range* using the function **CONSTRUCT** given in Alg. 4.3.

The algorithm uses the array `range` to represent the current reconstruction of the object and when a new view is appended to `P`, the `range` is modified after incorporating the information from the new silhouette. Throughout the algorithm, the front face is considered as the reference and each voxel of the object is represented with reference to this face.

### 5.2.1 UPDATING THE VI-hull

At any stage during the process, the VI-hull is primarily the collection of `range` arrays with respect to each black pixel of the FRONT and when a new silhouette is obtained the VI-hull is updated accordingly. As described earlier, it is assumed that the new silhouette is obtained in a direction which lies in one of the three coordinate planes with reference to the front face. The three cases are studied separately when the viewing direction lies in each of the three planes.

Let  $A$  be the binary matrix of size  $N \times N$  with respect to the silhouette in the direction  $a$  which lies in front-side plane with an angle  $\theta$  to the front viewing direction. Since silhouettes taken from  $\theta$  and  $180+\theta$  are identical, it is assumed that  $\theta$  lies between 0 and 180. For each row  $i$  of  $A$ ,  $1 \leq i \leq N$ , let  $m_1$  and  $m_2$  be a pair of integers as defined for SIDE. Then the `range` array is updated. A detailed algorithm is given in Alg. 5.1.

```

-----
function front_side (viewfs,  $\theta$  )
/* viewfs is the silhouette of size  $N \times (N/\cos\theta)$  taken from an angle of  $\theta$ 
to the front */

begin

  for i = 1 to N step 1
    begin

      compute  $m_1$  and  $m_2$  for row of viewfs;
      for j = 1 to  $N/\cos\theta$  step 1

        if (  $0 < \theta < \pi/2$  ) then

          range[i,j,0] = min ( range[i,j,0], ( $m_1 - (j+1) \cos \theta$ ) /  $\sin \theta$  );
          range[i,j,1] = max ( range[i,j,1], ( $m_2 - j \cos \theta$ ) /  $\sin \theta$  );

        else

          range[i,j,0] = min ( range[i,j,0], ( $m_1 - j \cos \theta$ ) /  $\sin \theta$  );
          range[i,j,1] = max ( range[i,j,1], ( $m_2 - (j+1) \cos \theta$ ) /  $\sin \theta$  );
        end;

      end;
    end.
  end.
-----

```

### Algorithm 5.1

It is to be noted that range may take on non-integer values too. Such non-integer values are considered so long as no difficulty arises.

If  $a$  lies in front-top plane then the VI-hull can be updated in a similar manner with the following changes. A pair of integers  $k_2$  and  $k_1$  are determined for each column of  $A$  and update the array range. The procedure is presented in Alg. 5.2.

```

-----
function front_top (viewft,  $\theta$  )
/* viewft is the silhouette of size  $(N/\cos\theta) \times N$  taken from an angle of  $\theta$  to
the front */

begin

  for j = 1 to  $N/\cos\theta$  step 1
    begin

```



```

compute  $k_2$  and  $k_1$  for column  $j$  of viewft;
for  $i = 1$  to  $N$  step 1

```

```

    if (  $0 < \theta < \pi/2$  ) then

```

```

        range[i,j,0] = min ( range[i,j,0], ( $k_1 - (i+1) \cos \theta$ ) /  $\sin \theta$  );
        range[i,j,1] = max ( range[i,j,1], ( $k_2 - i \cos \theta$ ) /  $\sin \theta$  );

```

```

    else

```

```

        range[i,j,0] = min ( range[i,j,0], ( $k_1 - i \cos \theta$ ) /  $\sin \theta$  );
        range[i,j,1] = max ( range[i,j,1], ( $k_2 - (i+1) \cos \theta$ ) /  $\sin \theta - 1$  );
    end;

```

```

end;

```

```

end.

```

---

### Algorithm 5.2

If the direction  $\mathbf{a}$  lies on the plane top-side then the range is updated as given in Alg. 5.3.

---

```

function top_side (views,  $\theta$  )

```

```

begin

```

```

    for  $i = 1$  to  $N$  step 1
        for  $j = 1$  to  $N$  step 1

```

```

             $p_{ij} = \text{int}( i \cos \theta + j \sin \theta )$ 

```

```

            range[i,j,0] = min ( range[i,j,0],  $m_1$  at row  $p_{ij}$  of  $A$  );

```

```

            range[i,j,1] = max ( range[i,j,1],  $m_2$  at row  $p_{ij}$  of  $A$  );

```

```

        end.
    end.

```

---

### Algorithm 5.3

#### 5.2.2 FIXING A VOXEL

The VI-hull of an object contains the object as a subset. This is due to the fact that we attempt to extract as much complete information (3D construction) as possible from the partial information (silhouettes). The correspondence between the

voxels of the actual object and the pixels of any silhouette is many-one and the volume intersection algorithm assigns an attribute (either black or, white) to each of the voxels of the object frame based on the similar attribute information available for the corresponding pixels in the silhouettes. If there is a single evidence that the voxel is white, then ignoring all other evidences that it may be black, the voxel is assigned the attribute white. Similarly a voxel is assigned the attribute black only when each of the evidences asserts that the voxel is black. When this principle is followed the following is clearly observed.

(i) Every voxel which is assigned 'white' during the algorithm is actually white in the object frame.

(ii) The voxel, which is assigned 'black', may or, may not be actually black in the object frame. But, however, at least some percentage of such voxels are surely black in the original object.

This is the consequence of the many-one relationships among the silhouettes and the object. The correspondence between the objects and the silhouettes is many-one. i.e., many different objects can give raise to the same silhouette. In a universe of side  $N$ , if a pixel is black in the silhouette, then it could be black because of any one of the voxels that project onto that pixel being black. Further, many combinations of runlengths of black and white voxels may result in the pixel being black. For the non-convex objects and objects with holes, the number of combinations of runlengths of black and white voxels is still

high. Let  $\phi(m)$  denote the number of different combinations of runlengths of black and white voxels in a total of  $m$  voxels.

Given an integer,  $m$ , we say that the sequence of integers  $m_1, m_2, \dots, m_r, m_1 \leq m_2 \leq \dots \leq m_r$  constitute a partition of  $m$  if  $m = m_1 + m_2 + \dots + m_r$  [Herstien 78]. Let  $p(m)$  denote the number of partitions of  $m$ . We can consider a partition of  $N$  to represent a combination of runlengths of white and black voxels, with the convention that the first number in the partition denotes the runlength of white voxels, the second that of black voxels, the third that of white voxels and so on.

But in  $\phi(m)$  the decompositions are not ordered - they can be increasing, decreasing or neither. Hence  $\phi(m) > p(m)$ . The function  $p(m)$  grows astronomically high as  $m$  increases. In fact,  $p(61) = 1,121,505$  [Herstien 78].

Thus, if there are  $b$  black pixels in the silhouette, the number of different objects that could give raise to the same silhouette is  $O(b \cdot \phi(N))$

Out of the many voxels corresponding to a single black pixel, at least one voxel is definitely black. But, the algorithm assigns each of such voxels to be black unless any contradiction is derived from other silhouette-information. Thus the VI-hull has some additional black voxels compared to the original object. By the process of 'Fixing a Voxel', we mean that whether we can decisively say if a voxel with assigned attribute as black is actually a black voxel in the original object. In other words, we attempt to derive some more information from the available evidences by not only assigning unambiguous attribute to white voxels but also eliminating ambiguity from some of the black

voxels. This process helps us on two occasions.

Firstly, all the unambiguous voxels are the maximal set of voxels carrying complete information. Hence, a huge class of 3D objects can be reconstructed by alternating the attribute assignment of ambiguous voxels without affecting the silhouette-informations. Thus the ratio of unambiguous voxels to the total number of voxels of the object frame provides a measure of accuracy of the reconstruction.

Secondly, a new direction of view provides one additional information which is used to convert some of the ambiguous-black voxels to the unambiguous-white. Thus if the knowledge of all the unambiguous voxels is available then one can intelligently choose an additional view so that the maximum number of ambiguous black voxels are fixed in the subsequent step and the redundant information regarding unambiguous voxels is avoided.

There can be many complicated rules for fixing any voxel which can involve computationally intensive algorithms. However, only two simple and efficient schemes are followed here. The main idea behind these schemes is to identify the many-one relations which are indeed one-one.

While updating the VI-hull using the `range` array, if for any black pixel  $(i,j)$  of `FRONT` the `range` contains only one voxel then we fix this voxel. Further if there is single voxel in any row or column of the 3D matrix '`range`', it is fixed.

If the binary array `A`, corresponding to the direction `a` on the front-side plane, contains a single black pixel along any column then all black voxels of the current reconstruction

corresponding to this pixel are fixed. Similarly for a direction on the front-top plane the black voxels corresponding to the lone black pixel in any row are fixed. When the new silhouette is taken in a direction on top-side plane, the black voxels of the current reconstruction are fixed for solitary black pixel in any row or, column. See Alg. 5.4.

```

-----
function fix_lone()

begin

  for i = 1 to N step 1
    for j = 1 to N step 1

      begin

        if (FRONT[i,j] is black and range[i,j,0]=range[i,j,1]) then
          fix voxel at (i,j,0) and break;

        if ( range[i,j,0] is the unique minimum of the ranges in row i )
          then fix voxel at (i,j,0);

        if ( range[i,j,1] is the unique maximum of the ranges in row i )
          then fix the voxel at (i,j,1) and break;

        if ( range[i,j,0] is the unique minimum of the ranges in column j )
          then fix the voxel at (i,j,0);

        if ( range[i,j,1] is the unique maximum of the ranges in column j )
          then fix the voxel at (i,j,1) and break;

      end;

    end.

  end.
-----

```

#### Algorithm 5.4

It can be noted here that the process of fixing a voxel does not require any extra computational efforts. The process of identifying the single black pixel along a row or a column can be carried out while the binary array is scanned during the updating of the VI-hull. But, however, the proposed method of fixing a voxel, fixes only boundary voxels.

### 5.3 ACCURACY MEASURE

To assess the performance of the algorithm when the input is a set of silhouettes taken in arbitrary viewing directions, it is essential to define a method to quantify the accuracy with which the volume of an object is approximated by the volume intersection algorithm. But, defining a measure of accuracy is not easy, for the approximation depends on the number of viewing directions, the nature of the object and the direction of views.

Let  $O$  be an object and  $O^*$  be the reconstructed approximation. It is natural to define the relative error of any volume intersection algorithm as

$$\frac{\text{vol}(O^*) - \text{vol}(O)}{\text{vol}(O^*)}$$

But, to measure the volume of the object  $O$ , in general, is not trivial. Further, unless the object is known a priori,  $\text{vol}(O)$  can not be determined.

Albeit,  $\text{vol}(O)$  is difficult to determine,  $\text{vol}(\bar{O})$  is not. Besides the object  $O$ , there could be several other objects  $O_1, O_2, \dots$  whose approximation is  $O^*$  too. We define  $\bar{O}$  as

$$\bar{O} = \{ \min \text{vol}(O_i) \mid O_i^* = O^* \}$$

The volume intersection algorithm generates the same  $O^*$  for all such objects  $O_1, O_2, \dots$ . However,  $\bar{O}$  is one such with minimum volume. Now define the worst-case relative error as

$$\frac{\text{vol}(O^*) - \text{vol}(\bar{O})}{\text{vol}(O^*)}$$

But, since the objects and their approximations are made up of the basic building blocks, voxels, and our volume intersection algorithm has complete information about each of the individual

voxels, the measure of accuracy is defined below ,  $\Delta$ , in terms of voxels.

By the process Fixing a Voxel, we ascertain that there is no ambiguity associated with a fixed voxel. Thus if the VI-hull contains all the fixed voxels and no other voxels, then we conclude that the best possible reconstruction is achieved. However it is not possible to eliminate the ambiguity of interior voxels by the method proposed here. Moreover, assuming that no scheme of volume intersection can acquire information related to cavities and holes, it is necessary to drop the ambiguity from all the interior voxels at this stage for the final reconstruction and measure the accuracy of reconstruction. The following scheme is proposed for such process.

If two voxels are fixed then we also fix all other black voxels which lie in the same line of view with these two voxels.

The implementation of above rule involves intricate integer arithmetic as it is difficult to find voxels which lie completely along the line of sight. A simple implementation is possible if the line of sight is restricted to be one of the three orthogonal views as front, top or, side.

- (i) If the voxels corresponding to two ends of the range are fixed then the whole range may be fixed.
- (ii) Every range must have at least one voxel fixed.

We define the measure of accuracy as follows;

$$\Delta = \frac{bf + w}{N^3}$$

where  $bf$  = total number of black voxels that are fixed,

$W$  = total number of white voxels,

and  $N$  = size of the array, hence  $N^3$  is the total number of voxels.

When  $\Delta$  is very close to 1, the algorithm is terminated and otherwise, if the accuracy of the reconstruction is not satisfactory then a new direction is determined after unfixing the voxels at the previous process.

### 5.3.1 DETERMINATION OF THE NEW DIRECTION

In this section a heuristic is proposed to determine the new direction of view. Suppose two voxels  $v_1$  and  $v_2$  are fixed in the process of 'Fixing a Voxel' along the directions  $D_1$  and  $D_2$ , respectively, such that  $D_1 \neq D_2$  and  $D_1, D_2 \in P$ . As discussed earlier, these voxels are black in the original object without any ambiguity. The voxels that lie 'behind' the fixed voxels cannot be resolved for ambiguity. Hence, in such cases, any new direction should aim at 'seeing' any critical voxels that are not obstructed by any of these fixed voxels. Thus, the direction in which  $v_1$  and  $v_2$  are collinear is a candidate for being the new entry to  $P$  provided all other fixed voxels of the object lie only to one side of this line of view. Such a restriction enables the new direction to separate the current approximation into two regions, one containing all the fixed voxels, presumably the main part of the object, and the other that may have ambiguous voxels. The direction which has the maximum number of unfixed voxels on the other side is taken as the new direction of view to be appended to  $P$ . The process to determine whether the whole set of fixed voxels lies to one side requires computational time proportional to the size of the array,  $N$ . However, determination



of the best viewing angle among the candidate set of directions is an intricate process. The mathematical formulation to find the best new direction involves certain algorithms in computational geometry, such as convex hulls and certain discrete optimization problems which are intractable. Hence, we propose a heuristic and we choose that candidate direction that is the nearest to one of the edge-on views.

#### 5.4 GENERATION OF LINEAR OCTREE

If the accuracy of the reconstruction is acceptable then no further silhouette is required for the refinement. However, from the intermediate data structure *range*, we generate the final representation of the 3D object in the form of a linear octree. This process of generating a linear octree from *range* with FRONT as the reference, has been described in the previous chapter. At this stage we take the *range* values to be integers, i.e., the floor of  $range[i,j,0]$  and the roof of  $range[i,j,1]$ .

#### 5.5 CONCLUSIONS

In this chapter, a method is proposed to reconstruct the 3D object from a set of 2D silhouettes. The 2D silhouettes are represented as binary arrays and the 3D object is represented as a linear octree. Unlike earlier work on this area, here an incremental technique is presented. From the initial reconstruction the algorithm tries to determine a new direction so that the subsequent reconstruction is an improvement over the earlier one. Such a process demands for a volume intersection algorithm which takes into account the silhouette in any

direction. We present a very simple scheme for considering any viewing direction by defining an intermediate data structure, and a scheme to measure the accuracy of the reconstruction resulted from the algorithm.

# CHAPTER 6

## Conclusions

The present work dealt with the study of some of the representation schemes of 2D and 3D objects. It consisted of two parts.

In the first part, a data structure is proposed for storing 2D polygons. The data structure, called the Edge k-d tree, is a balanced binary tree of height of  $O(\log n)$ , where  $n$  is the number of vertices. The edge k-d tree is constructed from a cyclic list of vertices of the polygon by recursively partitioning the set of vertices by horizontal and vertical cuts, alternately, along the median vertex sorted on  $X$  and  $Y$  coordinates respectively. The medians are stored in the internal nodes and the equations of edges in the leaf nodes.

The algorithm for constructing the edge k-d tree takes  $O(n^2)$  space and  $O(n \log^2 n)$  time for constructing the edge k-d tree, and it takes  $O(\log n)$  time to answer the point membership query. This is basically an ordered tree traversal of the edge k-d tree, to the edge that determines if the point is inside or outside the polygon. The operations of translation and scaling are tree traversals, making the appropriate modifications in the nodes. Hence, if there are  $m$  nodes in the tree, it takes  $O(m)$  time to reconstruct the edge k-d trees for the translated and scaled polygons.

The edge k-d tree is a translation and scaling invariant

data structure and thus it is useful for object recognition tasks. In a Geographic Information System, a frequently asked query is regarding the point membership problem and it takes only  $O(\log n)$  time to answer the query using the edge k-d tree.

In the second part, we presented a method of reconstructing the linear octree from a set,  $P$ , of silhouettes, using the volume intersection technique. Initially  $P$  contains three mutually orthogonal silhouettes, viz, front, side, and top. The object is reconstructed by intersecting the swept volumes obtained by sweeping the silhouettes along the respective viewing directions. But the object reconstructed through the volume intersection is only an approximation. A method is presented to gradually refine the approximation by appending  $P$  with silhouettes taken from additional viewing directions. A heuristic is presented to choose the next direction of view. A measure is also defined to determine the accuracy of the reconstructed object. The measure is defined as the ratio of the number of unambiguous voxels to that of the total number of voxels. Thus, the measure of accuracy is 1 for a reconstruction that is identical to the original object. We keep taking additional silhouettes to refine the approximation till the desired level of accuracy is obtained.

For the construction of the linear octree, an intermediate data structure, an array `range` is used. As and when new silhouettes are obtained, the array `range` is updated and finally, the locational codes of the octants are generated. The time complexity of generating the linear octree is of  $O(N^2R)$ , where  $N$  is the size of the universe and  $R$ , the number of octants in the linear octree.

Some aspects that need to be looked into further are :

(i) We presented an algorithm to build an edge k-d tree from a list of vertices of the polygon. The edge k-d tree answers the point membership problem in  $O(\log n)$  time and is translation and scaling invariant. We would like to look into the set operations on edge k-d trees, i.e., that of intersection, union and set difference of two polygons represented by their edge k-d trees.

(ii) A 3D object is said to be representable if there is a finite set of viewing directions such that the intersection of the swept volumes obtained by sweeping the silhouettes is congruent to the original object [Veenstra and Ahuja 85]. Thus, a parallelepiped is representable whereas a sphere is not. Also, objects that have holes or cavities are not representable. One is interested in knowing the kind of objects that are representable. In terms of the more elementary constructs like points and lines, the set of representable objects can be characterized. Veenstra and Ahuja [85] assert that an object is representable if there is a finite set of viewing directions such that for every point  $p$  outside the object, there exists a line parallel to a viewing direction which contains  $p$  and does not intersect the object.

Although such a definition makes it possible for the human eye to determine if an object is representable by looking at it, the machine may not do so. One would like to have a method for making the machine to decide the kind of objects that are representable in an acceptable time.

(iii) In the method proposed to reconstruct the 3D object by the volume intersection algorithm, we clarify the ambiguity regarding the color of a voxel by 'fixing' it. Here the voxels are fixed, a voxel at a time. As  $N$  increases, the total number of voxels, i.e.,  $N^3$  increases geometrically. As such, fixing of voxels may take more time. In such a case, we would like to see if we can 'fix octants'. In an octree set up, we are interested in exploring the possibility of fixing octants, i.e., to see if the color of the octants can be determined definitely.

(iv) When the volume intersection technique is used for reconstructing objects with holes or cavities, they are 'filled' up in the reconstruction. We would like to explore the possibility of generating such objects. In the present day, getting the silhouette of any cross section is not difficult (one can use the CAT (computer axial tomography) slices). Since the ordinary silhouettes do not give any information regarding the interior of an object and a cross sectional silhouette can, making use of both the kinds, one would like to get the best approximation. Yau and Srihari [83] present a method to reconstruct the octree from a series of cross sectional silhouettes. Using this extra information from the cross sectional silhouettes, the range can be updated and then the linear octree be generated.

(v) In our study, we restricted that the side and top silhouettes be monotonic along the  $X$  and  $Y$  axes respectively. Such a constraint will ensure the runlengths of black pixels in

the silhouettes being single intervals. For non-convex objects, where the runlengths could be an union of intervals, one can use a connected list of intervals. Here, each list of intervals represent the runlength corresponding to a row. We would like to investigate the use of segment trees [Wu et al 88] for storing runlengths and for generating the linear octree.

## R E F E R E N C E S

---

(Note : The papers are referred to by the Author(s) name and the year of publication. Since all the papers are published in this century only, the century index, '19', is omitted.)

|           |  |
|-----------|--|
| ACM-CS    | ACM Computing Surveys  |
| ACM-TG    | ACM Transactions on Graphics                                   |
| CACM      | Communications of the ACM                                      |
| CGIP      | Computer Graphics and Image Processing                         |
| CJ        | The Computer Journal   |
| CVGIP     | Computer Vision, Graphics, and Image Processing                |
| IEEE-PAMI | IEEE Transactions on Pattern Analysis and Machine Intelligence |
| IEEE-RA   | IEEE journal of Robotics and Automation                        |
| IEEE-SE   | IEEE Transactions on Software Engineering                      |
| IEEE-TC   | IEEE Transactions on Computers                                 |
| IPL       | Information Processing Letters                                 |
| JACM      | Journal of the ACM   |
| PRL       | Pattern Recognition Letters                                    |

Abel, D.J. 84. A B+ tree structure for large quadrees. CVGIP, 27, pp 19-31.

Abel, D.J. and Smith, J.L. 83. A data structure and algorithm based on a linear key for a rectangle retrieval problem. CVGIP, 24, 3, pp 1-13.

Aho, A.V., Hopcroft, J.E. and Ullman, J.D. 74. The design and analysis of computer algorithms. Addison Wesley. Reading (Mass).

Anedda, C. and Felician, L. 88. P-compressed quadrees for image storing. CJ, 31, 4, pp 353-357.

Atkinson, H.H., Gargantini, I. and Ramanath, M.V.S. 84. Determining the 3D border by repeated elimination of internal surfaces. Computing, 32, pp 279-295.

Atkinson, H.H., Gargantini, I. and Ramanath, M.V.S. 85b. Improvement to a recent 3D-border algorithm. PRL, 18, pp 215-226.

Atkinson, H.H., Gargantini, I. and Walsh, T.R.S. 85a Counting regions, holes and nesting level in time proportional to the border. CVGIP, 29, pp 196-215.

Ayala, D., Brunet, P., Juan, R. and Navazo, I. 85. Object representation by means of nonminimal division quadrees and octrees. ACM TG, 4, 1, pp 41-59.

Ballard, D.H. 81. Strip trees : A hierarchical representation for curves. CACM, 24, 5, pp 310-321.



Ballard,D.H. and Brown,C.M. 82. Computer Vision. Prentice Hall Inc, Englewood Cliffs, New Jersey.

Bauer,M.A. 85. Set operations on linear quadtrees. CVGIP, 29, pp 248-258.

Bentley,J.L. 75. Multidimensional binary search trees used for associative searching. CACM, 18, 9, pp 509-517.

Bentley,J.L. and Friedman,J.H. 79. Data structures for range searching. ACM CS, 11, 4, pp 397-409.

Bhaskar,S.K, Rosenfeld,A. and Wu,A.Y. 88. Parallel processing of regions represented by linear quadtrees. CVGIP, 42, pp 371-380.

Brunet,P. and Ayala,D. 87. Extended octtree representation of free form surfaces. Computer Aided Geometric Design, 4, pp 141-154.

Burton,F.W., Kollias,V.J. and Kollias,J.G. 87. A general PASCAL program for map overlay of quadtrees and related problems. CJ, 30, 4, pp 355-361.

Chen,H.H. and Huang,T.S. 88. A survey of construction and manipulation of octrees. CVGIP, 43, pp 409-431.

Chien,C.H. and Aggarwal,J.K. 84. A normalized quadtree representation. CVGIP, 26, pp 331-346.

Chien,C.H. and Aggarwal,J.K. 86a. Volume/Surface octrees for the representation of three dimensional objects. CVGIP, 36, pp 100-113.

Chien,C.H. and Aggarwal,J.K. 86b. Identification of 3D objects from multiple silhouettes using quadtrees/octrees. CVGIP, 36, pp 256-273.

Connolly,G.I. 84. Cumulative generation of octree models from range data. In Proc Intl Conf on Robotics, Atlanta, GA, pp 25-34.

Davis, L.S. and Russopoulos, N. 80. Approximate pattern matching in a pattern data base system. Information Systems, 5, 2, pp107-119.

Diehl,R. 88. Conversion of Boundary representation to Bintrees. Proc. of Eurographics 88. Ed Duce.D.A. and Jancene.P. North-Holland, pp 117-127.

Doctor,L.J. and Torborg,J.G. 81. Display techniques for octree-encoded images. IEEE Comp Graph Applns, 1, pp 29-40.

Dyer,C.R. 82. The space efficiency of quadtrees. CGIP, 19, pp 335-348.

- Dyer, C.R., Rosenfeld, A. and Samet, H. 80. Region representation : Boundary codes from quadrees. CACM, 23, 3, pp 171-179.
- Fabrini, F. and Montani, C. 86. Autumnal quadrees. CJ, 29, 5, pp 472-474.
- Finkel, R.A., and Bentley, J.L. 74. Quad trees, A data structure for retrieval on composite keys. Acta Informatica, 4, pp 1-9.
- Freeman, H. 74. Computer processing of line-drawing images. ACM CS, 6, 1, pp 57-97.
- Freeman, H. and Davis, L.S. 77. A corner finding algorithm for chain-encoded curves. IEEE-TC, 26, pp 297-303.
- Frieder, C.F, Gordon, D. and Reynolds, R.A. 85. Back-to-front display of voxel-based objects. IEEE Comp Graph Applns, 5, pp 52-60.
- Friedman, J.H., Bentley, J.L. and Finkel, R.A. 77. An algorithm for finding best matches in logarithmic expected time. ACM Trans Math Software, 3, pp 209-226.
- Gargantini, I. 82a. An efficient way to represent quadrees. CACM, 25, 12, pp 905-910.
- Gargantini, I. 82b. Detection of connectivity for regions represented by linear quadrees. Comp Math with Applns., 8, 4, pp 319-327.
- Gargantini, I. 82c. Linear octrees for fast processing of three dimensional objects. CGIP, 20, pp 356-374.
- Gargantini, I. 83. Translation, rotation, and superposition of linear quadrees. Intl J of Man-Mach stud., 18, pp 253-263.
- Gargantini, I. and Tabakman, Z. 82. Separation of connected components using linear quad- and oct-trees. Proc of 12th Conf on Num Math and Comp, Univ of Manitoba, Winnipeg, pp 257-276.
- Gibson, C.J. 83. A new method for the three-dimensional display of tomographic images. Phys Med Biol, 28, pp 1153-1157.
- Grosky, W.I. and Jain, R. 83. Optimal quadrees for image segments. IEEE-PAMI, 5, 1, pp 77-88.
- Herstien, I.N. 78. Topics in Algebra. Vikas Publishers.
- Hoare, C.A.R. 61. Partition, quicksort and find. CACM, 4, 7, pp 321-322.
- Hunter, G.M. 77. Computer animation survey. Computers and Graphics, 2, pp 225-229.

Hunter,G.M. 78. Efficient computation and data structures for graphics. Ph.D dissertation. Princeton University, Princeton, N.J.

Hunter,G.M. and Steiglitz,K. 79b. Linear transformations of pictures represented by quadtrees. CGIP, 10, pp 289-296.

Hunter,G.M. and Steiglitz,K. 79a. Operations on images using quadtrees. IEEE-PAMI, 1, 2, pp 145-153.

Iyengar,S. and Gadagkar,H. 88. Translation invariant data structure for 3-D binary images. PRL, 7, pp 313-318.

Jackins,C.L. and Tanimoto,S.L. 80. Oct-trees and their use in presenting three dimensional objects. CGIP, 14, pp 249-270.

Jackins,C.L. and Tanimoto,S.L. 83. Quad-trees, Oct-trees and K-tres - A generalized approach to recursive decomposition of Euclidean space. IEEE-PAMI, 5, pp 533-539.

Juan,R. 88. Boundary to constructive solid geometry :A step towards 3D conversion. Proc. of Eurographics 88. Ed Duce.,D.A. and Jancene.P., North-Holland, pp 129-139.

Kawaguchi,E. and Endo,T. 80. On a method of binary picture representation and its application to data compression. IEEE-PAMI, 2, 1, pp 27-35.

Kawaguchi,E., Endo.T. and Matsuyama,J. 83. Depth-first expressions viewed from digital picture processing. IEEE-PAMI, 5, 4, pp 373-384.

Kim,Y.C. and Aggarwal,J.K. 83. Rectangular coding for binary images. In Proc IEEE Conf on Comp Vision and Patt Recog, pp 108-113.

Kim,Y.C. and Aggarwal,J.K. 86. Rectangular Parellelepiped coding : A volumetric representation of three-dimensional objects. IEEE RA-2, 3, pp 127-134.

Klinger,A. 71. Pattern and Search statistics. In Optimizing methods in Statistics, J.S.Rustagi, Ed. Academic Press, New York, pp 303-337.

Klinger,A. and Dyer,C.R. 76. Experiments in picture representation using regular decomposition. CGIP, 5, 1, pp 68-105.

Knowlton,K. 80. Progressive transmission of grey-scale and binary pictures by simple,efficient and lossless encoding schemes. Proc. of IEEE, 68, 7, pp 885-896.

Knuth.D.E. 68. The art of Computer programming, vol 1 : Fundamental Algorithms. Reading (Mass), Addison Wesley.

- Knuth,D.E. 73. The art of computer programming, vol 3: Sorting and Searching, Reading (Mass), Addison Wesley
- Lavakusha, Arun K.Pujari. and P.G.Reddy. 86. Edge k-d trees. Proc of Soc of Infn Scientists. Pilani, pp 151-161.
- Lavakusha, Arun K.Pujari. and P.G.Reddy. 87. Polygonal representation by Edge k-d trees. Communicated to PRL.
- Lavakusha, Arun K.Pujari. and P.G.Reddy. 89a. Linear octrees by volume intersection. CVGIP, 45, 3, pp 371-379.
- Lavakusha, Arun K.Pujari. and P.G.Reddy. 89b.. Volume intersection algorithm with changing direction of views. Proc.of Intl Workshop on Indust Applns of Mach Intell and Vision (MIV 89), Tokyo, pp 309-314.
- Lee,Y.T., and Requicha,A.A.G. 82. Algorithms for computing the volume and other integral properties of solids II. A family of algorithms based on representation conversion and cellular approximation. CACM, 25, pp 642-650.
- Li,M., Grosky,W.I. and Jain,R. 82. Normalized quadtrees with respect to translations. CGIP, 20, 1, pp 72-81.
- Matsuyama,T., Hao,L.V., and Nagao,M.84. A file organization for geographic information system based on spatial proximity. CVGIP, 26, pp 303-318.
- Meagher,D. 80. Octree encoding : A new technique for the representation, manipulation and display of arbitrary 3D objects by computers. TR-IPL-80-111, Rensselaer Polytechnic Institute.
- Meagher,D. 82a. Geometric modelling using octree encoding. CGIP, 19, 2, pp 129-147.
- Meagher,D. 82b. Efficient synthetic image generation of arbitrary 3-D objects. In Proc of Pattern Recognition and Image Processing, Las Vegas, NV, pp 473-478.
- Merrill,R.D. 73. Representation of contours and regions for computer search. CACM, 16, 2, pp 69-82.
- Minsky,M. and Papert,S. 69. Perceptrons: An introduction to computational geometry, MIT press, Cambridge, MA.
- Murphy,O.J. and Selkow,S.M. 86. The efficiency of using k-d trees for finding nearest neighbour in discrete space. IPL, 23, pp 215-218.
- Negroponte,N. 77. Raster scan approaches to computer graphics.Computers and Graphics, 2, pp 179-193.

Newman,W.M. and Sproull,R.F. 84. Principles of interactive computer graphics. McGraw-Hill, New York.

Oliver,M.A. and Wiseman,N.E. 83a. Operations on quadtree encoded images. CJ, 26, 1, pp 83-91.

Oliver,M.A. and Wiseman,N.E. 83b. Operations on quadtree leaves and related image areas. CJ, 26, 4, pp 375-380.

Oliver,M.A., King,T.R. and Wiseman,N.E. 84. Quadtree scan conversion. In Proc of Eurographics '84, pp 265-278.

Orenstein,J.A. 82. Multidimensional tries used for associative searching. IPL, 14, 4, pp 150-157.

Overmars,M.H. 83. The design of dynamic data structures. Lecture Notes in Comp Sc., Vol 156, Springer-Verlag, New York.

O'Rourke,J. and Badler,N. 77. Decomposition of 3D objects into spheres. IEEE-PAMI, 1, pp 295-305.

Pavlidis,T. 78. Survey : A review of algorithms for shape analysis. CGIP, 7, pp 243-258.

Peters,F.J. 85. An algorithm for transformation of pictures represented by quadtrees. CVGIP, 32, pp 397-403.

Pfaltz,J.L. and Rosenfeld,A. 67. Computer representation of planar regions by their skeletons. CACM, 10,2, pp 119-122 and 125.

Potemsil,M. 85. Generating octree models of 3-D objects from their silhouettes in a sequence of images. Technical Memo, AT & T Bell Labs.

Preparata,F.P. 88. Point-location problem - A guided tour of decade research. In Lecture notes in Computer Science, Springer-Verlag, New York.

Preparata,F.P. and Shamos,M.I. 85 Computational Geometry. Springer-Verlag, New York.

Reddy,D.R. and Rubin,S. 78. Representation of three dimensional objects. CMU-CS-78-113, Computer Science Department,Carnegie-Melon Univ, Pittsburg.

Requicha,A.A.G. 80. Representation of rigid solids : Theory, methods, and systems. ACM-CS, 12, 4, pp 437-464.

Requicha,A.A.G. and Voelcker,H.B. 80. A tutorial introduction to geometric modelling. SIGGRAPH 80 Tutorial.

Rosenfeld,A. 80. Three dimensional digital topology. TR-926, Computer Science Center, Univ of Maryland, College Park.

Rosenfeld, A. and Kak, A.C. 82. Digital picture processing. Academic Press, New York.

Rosenfeld, A. and Pfaltz, J.L. 66. Sequential operations in digital picture processing. JACM, 13, 4, pp 471-494.

Rutowitz, D. 68. Data structures for operations on digital images. In Pict Patt Recognition. G.C.Chang et al Eds. Thompson Book Co., Washington D.C. pp 105-133.

Samet, H. 80a. Region representation : quadrees from binary arrays. CGIP, 13, pp 88-93.

Samet, H. 80b. Region representation : quadrees from boundary codes. CACM, 23, 3, pp 163-170.

Samet, H. 81a. Computing perimeters of images represented by quadrees. IEEE-PAMI, 3, pp 683-687.

Samet, H. 81b. An algorithm for converting rasters to quadrees. IEEE-PAMI, 3, pp 93-95.

Samet, H. 81c. Connected component labelling using quadrees. JACM, 28, 3, pp 487-501.

Samet, H. 82a. Neighbor finding techniques for images represented by quadrees. CGIP, 18, pp 37-57.

Samet, H. 82b. Distance transform for images represented by quadrees. IEEE-PAMI, 4, 3, pp 298-303.

Samet, H. 83 A quadtree medial axis transform. CACM, 26, 9, pp 680-693.

Samet, H. 84a. The quadtree and related hierarchical data structures. ACM CS, 16, 2, pp 187-260.

Samet, H. 84b. Algorithms for the conversion of quadrees to rasters. CVGIP, 26, pp 1-16.

Samet, H. 85a. A top-down quadtree traversal algorithm. IEEE-PAMI, 7.

Samet, H. 85b. Data structures for quadtree approximation and compression. CACM, 28, 9, pp 973-993.

Samet, H. 85c. Reconstruction of quadrees from quadtree medial axis transform. CVGIP, 29, 2.

Samet, H. and Tamminen, M. 84. Efficient image component labelling. TR-1420, Computer Science Department, Univ of Maryland, College Park. MD.

- Samet, H. and Webber, R.E. 85. Storing a collection of polygons using quadtrees. ACM Trans on Graphics, 4, 3, pp 182-222.
- Scott, D.S. and Iyengar, S. 84. TID - A translation invariant data structure for storing images. TR 84-027, Dept of Comp Sci., Univ of Texas at Austin.
- Scott, D.S. and Iyengar, S. 86. TID - a translation invariant data structure for storing images. CACM, 29, 5, pp 418-429.
- Shaffer, C.A. and Samet, H. 87. Optimal quadtree construction algorithms. CVGIP, 37, pp 402-419.
- Shapiro, L.G. 79. Data structures for picture processing : A survey. CGIP, 11, pp 162-184.
- Shneier, M. 81a. Two hierarchical linear feature representations : Edge pyramids and edge quadtrees. CGIP, 17, 3, pp 211-224.
- Shneier, M. 81b. Calculation of geometric properties using quadtrees. CGIP, 16, pp 296-302.
- Simmons, G.F. 63. Introduction to topology and modern analysis, New York, Mc Graw-hill.
- Soroka, B.I. 79. Understanding objects from slices. Ph.D dissertation, Dept of Computer and Information Science, Univ of Pennsylvania.
- Srihari, S.N. 80. Hierarchical representation for serial section images. Proc of 5th Intl Conf on Pattern Recognition, Miami Beach, Florida, pp 1075-1080.
- Srihari, S.N. 81. Representation of three dimensional digital images. ACM CS, 13, 4, pp 399-424.
- Stewart, I.P. 86. Quadtrees : storage and scan conversion. CJ, 29, 1, pp 60-75.
- Tamminen, M. and Samet, H. 84. Efficient octree conversion by connectivity labelling. Proc of SIGGRAPH 84 Conf, Minneapolis, Minn, pp 43-51.
- Tang, Z. and Lu, S. 88. A new algorithm for converting boundary representation to octree. Proc. of Eurographics 88. Ed Duce, D.A. and Jancene, P. North-Holland, pp 105-116.
- Tanimoto, S. 76. Pictorial feature distortion in a pyramid. CGIP, 5, pp 333-352.
- Tanimoto, S. and Pavlidis, T. 74. A hierarchical data structure for picture processing. CGIP, 4, pp 104-119.



Unnikrishnan,A., Shankar,P. and Venkatesh.Y.V. 88. Threaded linear hierarchical quadrees for computation of geometric properties of binary images. IEEE-SE, 14, 5, pp 659-665.

Unnikrishnan,A., Venkatesh,Y.V. and Shankar,P. 87. Connected component labelling using quadrees - A bottom-up approach. CJ, 30, 2, pp 176-182.

Van Lierop,M.L.P. 86. Geometrical transformations on pictures represented by leafcodes. CVGIP, 33, pp 81-98.

Veenstra,J. and Ahuja,N. 85. Deriving object octree from images. In Proc of FST & TCS. Ed S.N.Maheswari, Lecture Notes in Comp Sc, No. 206, Springer-Verlag.

Veenstra,J. and Ahuja,N. 86. Efficient octree generation from silhouettes. Proc of Computer Vision and Pattern Recognition, Miami, FL, pp 537-542.

Voelcker,H.B. and Requicha,A.A.G. 77. Geometric modelling of mechanical parts and processes. Computer, 10, pp 48-57.

Walsh,T.R. 88. Efficient axis-translation of binary digital pictures by blocks in linear quadtree representation. CVGIP, 41, pp 282-292.

Warnock,J.L. 69. A hidden surface algorithm for computer generated half-tone pictures. TR 4-15, Comp Sc Deptt, Univ of Utah, Salt Lake City.

Weng,J. and Ahuja,N. 87. Octrees of objects in arbitrary motion: Representation and Efficiency. CVGIP, 39, pp 167-185.

Williams,R. 88. The goblin quadtree. CJ, 31, 4, pp 358-363.

Woodwark,J.R. 82. The explicit quadtree as a structure for computer graphics. CJ, 25, 2, pp 235-238.

Wu,A.Y., Bhaskar,S.K. and Rosenfeld,A. 88. Parallel computation of geometric properties from medial axis transform. CVGIP, 41, 3, pp 323-332.

Yamaguchi,K., Kunii,T.L. and Fujimara,K. 84. Octree-related data structures and algorithms. IEEE Computer Graphics Applications 4, pp 53-59.

Yau,M. and Srihari,S. 83. A hierarchical data structure for multidimensional digital images. CACM, 26,7, pp 504-515.