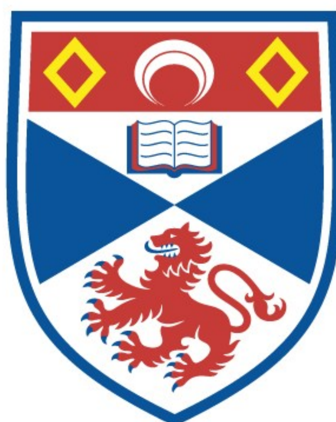
MURRAY POLYGONS AS A TOOL IN IMAGE PROCESSING

Bhuwan Pharasi

A Thesis Submitted for the Degree of PhD
at the
University of St Andrews



1990

Full metadata for this item is available in
St Andrews Research Repository
at:
http://research-repository.st-andrews.ac.uk/

Please use this identifier to cite or link to this item:
http://hdl.handle.net/10023/13580

This item is protected by original copyright

L(6(4-2-(

Murray Polygons as a Tool in Image Processing

thesis submitted
in fulfilment for the requirement of
the degree of
DOCTOR OF PHILOSOPHY

by

Bhuwan Pharasi.

Department of Computational Sciences,
University of St. Andrews
St. Andrews

October 1989

ProQuest Number: 10166334

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

ProQuest 10166334

Published by ProQuest LLC (2017). Copyright of the Dissertation is held by the Author.

All rights reserved.
This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Th A1074

To my mother Smt. Sumitra Devi Pharasi
and my father Sh. Mangla Nand Pharasi
also to my sister-in-law Mrs. Sheela Pharasi.
and to my brother Mr. Harsh V. Pharasi

I Bhuwan Pharasi hereby certify that this thesis has been composed by myself, that it is a record of my own work, and that it has not been accepted in partial or complete fulfilment of any other degree or professional qualification.

Signed .... Date 10/10/89

I was admitted to the Faculty of Science of the University of St.Andrews under Ordinance General No 12 on October 10,1986 and as a candidate for the degree of Ph.D on October 10, 1987.

Signed .... Date 10/10/89

I hereby certify that the candidate has fulfilled the conditions of the Resolutions and Regulations appropriate to the degree of Ph.D.

Signature of Supervisor .... Date 10/10/89

Copyright

In submitting this thesis to the University of St. Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker.

Acknowledgements

It gives me an immence pleasure and great opportunity to express my profound sense of gratitude and indebtedness to Professor A. J. Cole, for his painstaking guidance, invaluable suggestions and constant encouragement throughout the research work.

I wish to express my thanks to Professor R. Morrison; Chairman, for providing all necessary facilities and also to Dr. J. Owczarczyk for proof-read and helping me during this project.

I also wish to record my appreciation and feeling of gratitude to Mr. A. J. T. Davie, Dr. A. Brown, Dr. R. Dyckhoff, Mrs. H. Bremner, Mrs. E. Nicoll and Mr. B. McAndie who helped me in every possible way.

My humble regards are also due to my others family members who have always encouraged and assisted me to pursue higher studies.

I am also thankful to: the Committee of Vice-Chancellors and Principals, and St. Andrews University, for providing me the financial support.

(Bhuwan Pharasi)

Abstract

This thesis reports on some applications of murray polygons, which are a generalization of space filling curves and of Peano polygons in particular, to process digital image data. Murray techniques have been used on 2-dimensional and 3-dimensional images, which are in cartesian/polar co-ordinates. Attempts have been made to resolve many associated aspects of image processing, such as connected components labelling, hidden surface removal, scaling, shading, set operations, smoothing, superimposition of images, and scan conversion.

Initially different techniques which involve quadtree, octree, and linear run length encoding, for processing images are reviewed. Several image processing problems which are solved using different techniques are described in detail. The steps of the development from Peano polygons via multiple radix arithmetic to murray polygons is described. The outline of a software implementation of the basic and fast algorithms are given and some hints for a hardware implementation are described

The application of murray polygons to scan arbitrary images is explained. The use of murray run length encodings to resolve some image processing problems is described. The problem of finding connected components, scaling an image, hidden surface removal, shading, set operations, superimposition of images, and scan conversion are discussed. Most of the operations described in this work are on murray run lengths. Some operations on the images themselves are explained.

The results obtained by using murray scan techniques are compared with those obtained by using standard methods such as linear scans,

quadtrees, and octrees. All the algorithms obtained using murray scan techniques are finally presented in a menu format work bench. Algorithms are coded in PS-algol and the C language.

CONTENTS

INTRODUCTION x

1. REPRESENTATION AND EXACT COMPRESSION OF DIGITAL IMAGES 1

1 - 1 Introduction 1

1 - 2 Run-Length Encoding 3

Linear Scan 5

Space Filling Curves 6

1 - 3 Quadtree Encoding 9

Leaf Node 12

Traversal Of The Node Of Its Quadtree 15

1 - 4 Volume Data 16

2. MURRAY POLYGONS 20

2 - 1 Introdution 20

2 - 2 Murray Polygons 20

Gray Codes or Cyclic Progressive Numbers 23

Direct Peano Transformations 25

Murray Arithmetic 26

Murray Transformation 28

Mixed Scan 31

Some Lemmas 32

Implementation Of Murray Scan 39

Original Implementation 39

A Faster Murray Scan Algorithm 42

Hardware Implementation 46

Three-Dimensional Cartesian Coordinates 46

Extension To 3-D And n-D Murray Polygons 47

Method One 47

Some Lemmas 50

Second Method 52

Polar Murray Scan 54

Polar Coordinates 54

Changing Coordinates Systems 55

Graphs In Polar Coordinates 56

3D And Higher Dimensional Polar Coordinates 56

Cylindrical And Spherical Coordinates 56
Implementation Of Planar Polar Murray Scan 59
Cylindrical Polar Murray Scan 61
Spherical Polar Murray Scan 61
2-2.11 Application Areas 62
Scanning 62
2-2.12 Remarks 63

3. SCANNING AND DRAWING OF THE IMAGES 64
3-1 Introduction 64
3-2 Structure And List Processing 65
3-3 Linked List 68
3-4 Image Construction 69
3-5 Storing an Image in a Database 72
Retrieving an Image From a Database 74
3-6 Scanning And Drawing Of An Image 74
3-7 Remarks 80

4. SCAN CONVERSION AND SCALING OF IMAGES 83
4-1 Introduction 83
4-2 Scan Conversion 84
Method 1 84
Some Lemmas 89
Method 2 96
Some Lemmas 98
4-3 Implementation of Scan Conversion 104
Data Structure 104
Scanning 107
Algorithms 108
Comparison Between The Two Algorithms 112
Comparison Between Linear and General Murray Scan 113
4-4 Scaling 114
Introduction 114
Scaling Using Murray Polygons And Its Implementation 116
Some Lemmas 122
Theorem 123
Results 132
4-5 Remarks 134

5. SUPERIMPOSITION, AND SET OPERATIONS ON IMAGES 137

5-1 Introduction 137

5 - 2 Set Operations 138

5-3 Superimposition of The Images Using Murray Polygons 140

Implementation 141

5 - 4 Set Operations Using Murray Polygons 143

Union 144

Intersection 146

Difference 146

5 - 4 Remarks 147

6. CONNECTED COMPONENET LABELLING 149

6 - 1 Introduction 149

6 - 2 Connected Component Labelling 151

6 - 3 Connected Component Labelling Using Murray Polygons 153

Method 1(Using Images) 155

Method 2 162

Using Two Sequences of Runlengths 162

Extension To 3-Dimensional and n-D Images 168

Comparison Between Method 1 And Method 2(part 1) 170

Using One Sequences of Runlengths 170

Extension To 3-Dimensional and n-D Images 178

Comparison Between Method 2 (Two list vs One list) 179

6 - 3 Remarks 179

7. HIDDEN SURFACE REMOVAL AND SHADING 182

7 - 1 Introduction 182

7 - 2 Hidden-Surface Removal 183

Object-Space Algorithms 184

Image-Space Algorithms 186

List-Priority Algorithms 186

Scan Line Algorithms 195

Scan Line Coherence Algorithms 196

A Visible Surface Ray Tracing Algorithm 197

Octree Methods 198

7 - 3 Hidden-Surface Removal Using Murray Polygons 199

Method 1 201

Method 2 202
Comparison Of Hidden Surface Methods 209
7-4 Shading 210
Introduction 210
Surface Shading Methods 212
Transparency 216
Texture Mapping 219
Antialiasing 219
Shadows 220
7 - 5 Shading Using Murray Polygons 221
Determining The Surface Normal 222
Determining The Intensity Using Murray Polygons 223
Determination Of The Angle Between N And L 225
Smoothing Of Data 232
Results 235
Conclusion 239
7-6 Specular Reflection 240
7 - 5 Remarks 242

8. IMPLEMENTATION 244
8-1 User Interface 244
8-2 Menu Design 245

9. CONCLUSIONS AND FUTURE WORK 252
9 - 1 Conclusions 252
9 - 2 Future Work 256

REFERENCES 257

INTRODUCTION

The use of image processing is increasing and is being widely used in many industries. Medicine, meteorology, mapping, industrial vision, publishing, and television are just few of the applications of modern image processing systems. In many of these cases image processing is helping to derive more information from the image data. For example, in meteorology much more information can be extracted from satellite pictures by processing the data as it is received. This information includes finding signs of mineral deposits, finding about enemy activities, weather information, etcetera. Further in the field of medicine image processing techniques can be used in extracting out information about the disease from the images which are obtained by the CT scanner. In industrial applications images can be used to identify whether a product is good or bad. In this case the images which are obtained by a vision capture system, usually a camera, can be compared with the stored image of a good component. If the image does not match with the one stored for a good product then it can be rejected.

Others common application areas are:

1. Animation/Graphic Arts;
2. Astronomy;
3. CAD/CAM/CAE;
4. Machine Vision;
5. Geographical/Environmental;
6. Storage and transmission of digital image data;
7. Simulation of various sort e.g., flight simulation., etcetera

The processing time and the storage or transmission capacity increases with the increase in the size of an image. Hence, there must be a method for encoding an image, which can reduce the amount of disk storage or transmission capacity, so as to be able to handle exact images and also to be able to carry out standard transformations on whole images or sub-images independently of and from the bit map itself. Various methods of recording the information in the bit maps for raster scan or bit mapped graphics(i.e., an image) have been suggested, the two most popular being linear run length encoding[Foley, and Van Dam(1982), Roger(1985), Hearn, and Baker(1986)] and quadtree or octree encoding[Klinger, and Dyer(1976), Samet(1984), Gargantini(1982)] Some investigations have also been made into the use of Hilbert scans[Hilbert(1891)] using table driven algorithms[Griffiths(1985), Cole(1985c)].

This thesis explains the use of murray polygons[Cole(1985b)] as a possible alternative to the above methods, in many related problems of image processing. Murray polygons are a generalisation of space filling curves and of Peano polygons[Peano(1890)] in particular. Many associated problems related to image processing are solved by using murray polygons and are compared with those already defined for linear or quadtree encoding.

The main characteristics of murray polygons are :

i. Instead of being restricted to squares, murray polygons may be defined in a variety of ways so as to pass through all points with integer coordinates in any rectangle with odd integer length sides. Murray polygons are not restricted to odd dimensions as the restriction on the radices being odd can be lifted for the first and the last radices giving even sided rectangles. This is discussed in the following chapters.

ii. Explicit transformation as well as recursive or table driven algorithms can be defined.

iii. By slight modification the same algorithm, which is defined for cartesian coordinates, may be used for polar coordinates.

iv. A murray linear scan has a minor advantage over a conventional linear scan. In a conventional linear scan, the flyback will usually result in a break of run length, which may result in more run lengths than that of a murray scan.

v. The distribution of murray run lengths is different to that of linear run lengths. This distribution may be exploited in a final coding of the run lengths for storage or transmission.

vi. Murray polygons can scan any rectangle of sides r and s, with no restriction on the values of r and s, in the horizontal as well as in the vertical direction. It is also possible to transform directly from a horizontal to a vertical murray scan and vice versa. This can be used when we have to scale the image in both directions, horizontally as well as vertically. More detailed discussion will be given in the following chapters.

vii. With a minor modification, the algorithms which are defined for 2-dimensional images can be used to scan 3-D and n-D images, with no restriction on the size of the images.

viii. Using a single 3D cartesian scan the whole image can be viewed in six possible directions(top, bottom, front, back, l-side, r-side).

ix. If an image is represented in spherical polar coordinates then it can be easily scanned from any given viewpoint.

x. Murray scans, although being locally n-dimensional in nature, still produce a linearised scan of the corresponding 3-D image. They also have the advantage of giving a choice of scanning order including the possibility of scanning in the colour code dimension rather than plane by plane.

Chapter 1, is a survey of image processing techniques, illustrating the diversity of the various methods which have been proposed by other authors. Murray polygons for 2D, 3D and higher order images are explained in chapter 2. Cartesian and polar murray scans are also explained in detail. Scanning and drawing images using murray polygons is explained in Chapter 3. Chapter 4 is about scan conversion and scaling images horizontally or vertically or in both directions. Chapter 5, explains about superimposition, and set operations on images. Chapter 6 explains about connected components labelling for images. Two methods are explained for identification of homogeneous connected components, either directly from the bit map or from a run length encoding. When a run length encoding is used the results themselves are recorded as runlength encodings. Hidden surface problems and shading techniques for 3D images are discussed in chapter 7. Some smoothing techniques are also discussed in chapter 7. Chapter 8 is about the work bench design and implementation. Initially all the algorithms were coded in PS-Algol; later on to improve on the speed for some algorithms the C language is used. Some fast software algorithms and a proposal for a hardware implementation which should enable a real time scan of a bit map to be made, are discussed. At the end of each chapter concluding remarks are given. Finally the results are summarized in the last chapter.

Chapter 1

1. REPRESENTATION AND EXACT COMPRESSION OF DIGITAL IMAGES 1

1 - 1 Introduction 1

1 - 2 Run-Length Encoding 3

Linear Scan 5

Space Filling Curves 6

1 - 3 Quadtree Encoding 9

Leaf Node 12

Traversal Of The Node Of Its Quadtree 15

1 - 4 Volume Data 16

1-1 Introduction :

According to Rosenfeld and Kak(1976) picture processing or image processing by computer surrounds a wide variety of techniques and mathematical tools. Most of these have been developed due to three major problems:

i. <u>Picture digitization and coding:</u> conversion of picture from continous to discrete form (digitization) and then coding the results so as to reduce the amount of storage space or transmission capacity.

ii. <u>Picture enhancement and restoration:</u> improvement of blurred (or noisy) pictures.

iii. <u>Picture segmentation and description:</u> conversion of pictures into simplified sub-pictures; classification or description of pictures in term of parts and properties.

<u>Picture :</u>

A picture is a flat object whose brightness or color may vary from point to point. For a black and white picture this can be mathematically represented by a single real-valued function, say f(x,y). The value of this function at a point will be called the *gray level* or *brightness* of the picture at that point. Further the values of this function are nonnegative and bounded, i.e., $0 <= f(x,y) <= M$ for all x, y.

<u>Pictures as Arrays:</u>

A digitized picture or digital picture, can be regarded as an integer array.The elements of a digital picture array are called picture elements, pixels, or pels. Once a picture has been digitized, additional processing

techniques can be applied to rearrange picture parts, to remove or process a large homogeneous area, to scale the picture up or down *etcetera*. In rest of this thesis we will refer to digital pictures as images.

The above definitions can now be summarised as: the term *image(or digital picture),* refers to the original array of pixels. If its elements are either BLACK or WHITE then it is said to be a binary image. If shades of gray are possible (i.e., gray levels), then the image is said to be a *gray-scale image.* A *pixel* is said to have four edges, each of which is of unit length.

An image represented by an $N*N$ square array of pixels, each of P bits, would need $P*N*N$ bits to store in uncoded form. In practice, it is found that neighbouring pixels are often the same and by using a suitable coding scheme, this one-or-two dimensional spatial coherence can be exploited to write the image in much less than $P*N*N$ bits. Various coding schemes have been used for processing the images. The two most popular coding schemes are

i. Run length encoding[Foley, and Van Dam(1982), Roger(1985), and Hearn, and Baker(1986)]

ii. Quadtree or octree encoding[Klinger, and Dyer(1976), Gargantini(1982) and Samet(1984)]

Many associated problems such as connectivity, scaling, merging superimposing, hidden surface removal, shading, etcetera, have been implemented using these methods.

In this chapter, different image processing methods and their use in exactly compressing the digital images data is explained. In each section, references will be provided as to where more detailed explanations can be found.

1-2 Run Length Encoding :

Initially the image is scanned to produce a sequence of run lengths. Each run can be encoded as a tuple (I_i,D_i), where D_i is the number of pixels, each with intensity I_i. D_i and I_i are usually stored as one byte each.

Intensity	Run Length

Intensity levels or gray scales, depend upon the number of bit planes per pixel. For N bit planes the intensity level will lie between 0 and 2^N-1, where 0 (corresponds to dark) and 2^N-1(corresponds to full intensity). A total of 2^N intensity levels can be achieved. If only one bit plane is provided in the raster, on (white) and off(black) are the only possibilities for the gray scale. Three bit planes per pixel can accommodate eight different intensity levels and so on. Many packages use the range 0 to 1 to set gray scale levels. Intensity values specified in a program are converted to appropriate binary codes for storage in the raster. Figure 1-2.1 illustrates conversion of user specification to codes for a four-level gray scale. In this example, any intensity input value near 0.33 would store the binary code 01 in the frame buffer and result in a dark gray shading for these pixels.

For black and white images (i.e., one bit per pixel), we generally assume that the first run length will always correspond to white pixels. If the first pixel is black then the first run length will be zero. Hence in the case of black and white images we do not require to store the intensity of the pixels.

The simple run length encoding scheme can easily be extended to include color. For color, the intensity of each of the red, green, and blue color guns is given followed by the number of successive pixels for that color e.g.,

Red Intensity	Green Intensity	Blue Intensity	Run Length

Usually the color intensities are combined into a single integer, referred as *color code,* using a fixed number of bits. A simple scheme for storing color code selections in the frame buffer of a raster system is shown in Figure 1-2.2. When a particular color code is specified in an application program, the corresponding binary value is stored in the frame buffer for each component pixel in the output primitives to be displayed in that color. The scheme is given in figure 1-2.2 allows eight color choices with 3 bits per pixel of storage. Each of the three bit positions is used to control the intensity level (either on or off) of the corresponding electron gun in an RGB monitor. The leftmost bit controls the red gun, the middle bit controls the green gun, and the rightmost bit controls the blue gun. Adding more bits per pixels to the frame buffer increases the number of color choices.

Run length coding can often substantially reduce the amount of memory needed to store images. Its advantage is maximised, in cases where the images are made up of a few long runs. To produce long run lengths, very much depends upon the *image* itself and upon the *scanning* methods. *According to Shannon's[Klerer, and Korn(1967)] information theory : On average to maintain complete information for all images one cannot do better than the bit map.*

INTENSITY CODES	STORED INTENSITY VALUES IN THE FRAME BUFFER (Binary Code)		DISPLAYED GRAY SCALE
0.0	0	(00)	Black
0.33	1	(01)	Dark Gray
0.67	2	(10)	Light Gray
1.0	3	(11)	White

Figure 1-2.1

Conversion of intensity values to integer codes for storage in a frame buffer accommodating a gray scale with four levels. Two bits of storage for each pixel position are needed in the frame buffer.

COLOUR CODE	STORED COLOR VALUES IN FRAME BUFFER			DISPLAYED COLOR
	RED	GREEN	BLUE	
0	0	0	0	Black
1	0	0	1	Blue
2	0	1	0	Green
3	0	1	1	Cyan
4	1	0	0	Red
5	1	0	1	Magenta
6	1	1	0	Yellow
7	1	1	1	White

Figure 1-2.2

Color codes stored in a frame buffer with three bits per pixels.

Run length encoding can further be divided into two classes which are,

i. Linear scans.

ii. Space filling curves.

1-2.1 Linear Scan [Netravali and Haskell(1988)]:

Linear scanning converts the two dimensional image intensity into a one dimensional waveform. The image is segmented into L_y adjacent horizontal lines, and the image is scanned one line at a time, sequentially, left to right, and top to bottom with fly back at the end of each scanline, see Figure 1-2.3. It is one dimensional in nature and takes advantage of the correlation between adjacent pixels on the scanline. For example, in a flying spot scanner a small spot of light scans across a photograph, and the reflected energy at any given position is a measure of the intensity at that point.

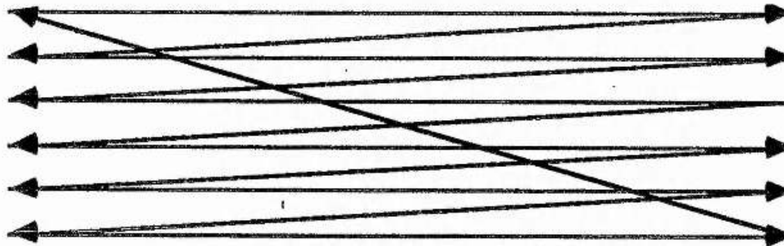
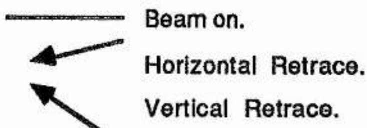


Figure 1-2.3



1-2.2 Space Filling Curve :

Giuseppe Peano(1890) introduced the idea of a space filling curve in the sense of a continuous mapping of the line segment (0,1) onto the unit square and was closely followed by Hilbert(1891) and Sierpinski(1912). Peano, introduced the idea of space filling curves rather than polygons and Hilbert and others introduced the idea of limiting sequences of polygons leading to space filling curves.

Peano showed how to produce a curve by moving a single point continuously over a square, such that it passes at least once through every point on the square and its boundary. The curve produced was indeed continuous but it was impossible to draw unique tangents, since it is impossible to specify the direction in which a points is moving. Two interesting points about the Peano curve were :

1) Its path seems to be one dimensional, yet at the limit it occupies a two dimensional area.

2) It is a continous curve, but has no derivative.

Peano based his definition of space filling curve on a base three representation of the points on the real line interval [0,1] and the points (x,y) of the square $0 <= x <= 1$, $0 <= y <= 1$. Essentially, a point of the above interval was split into two real base three numbers by taking all the odd indexed digits in their sequential order for the value of x and all the even ordered digits in their sequential order for the value of y. It should be noted that to maintain the uniqueness of the transformation, the ambiguity of real number representation, where to base three, 0.2211 and 0.221022222222........

are equally valid representations, was dealt with in a more complex manner, which need not be discussed in this report.

Hilbert generated a Peano curve with two end points. In other words, Hilbert derived an alternative method of defining a space filling curve as the limit of polygons enclosed in the unit square, using a fourfold repetition of successive polygons which corresponded to a base two number representation, as shown in Figure 1-2.5. At the limit Hilbert curve starts at the bottom left and finishes at the top left. A similar result based on the Peano technique was given by Moore(1900) to obtain a limiting polygon based on ninefold repetitions of successive polygons. These polygons are known as Peano Polygons. The first three Peano polygons P1,P2,P3 are shown in Figure 1-2.4.

The three steps of the illustration show how Waclaw Sierpinski generated a closed Peano curve (see Figure 1-2.6a). Sierpinski polygons differ from Hilbert and Peano polygons. The principal difference as Wirth[1976] pointed out is that Sierpinski curves are closed curves made up of four parts, which are connected by the four straight lines in the outermost four corners. Cole(1983) showed that S_n' can be obtained from S_{n-1}' by suitably rotating and shifting S_{n-1}' to four new positions and joining them by three lines. S_1', S_2', S_3' are shown in the Figure 1-2.6b. Further S_3 (see Figure 1-2.6a) is obtained by rotating S_3' (see Figure 1.2.6b)four times and finally closing the last gap. Wirth suggested that S_0 is a square standing on one corner. It means the starting curve is the single point [1,0]. Recursive algorithms for drawing these and other space filling curves have been given by Wirth(1976), Goldschlager(1981) and Witten and Wyvill (1983). Griffiths(1985) discusses table driven algorithms for generating space filling curves.

Helge Von Koch[Gardner(1967)], proposed in 1904 another curve which is now commonly called the snow flake curve. At the limit it is infinite

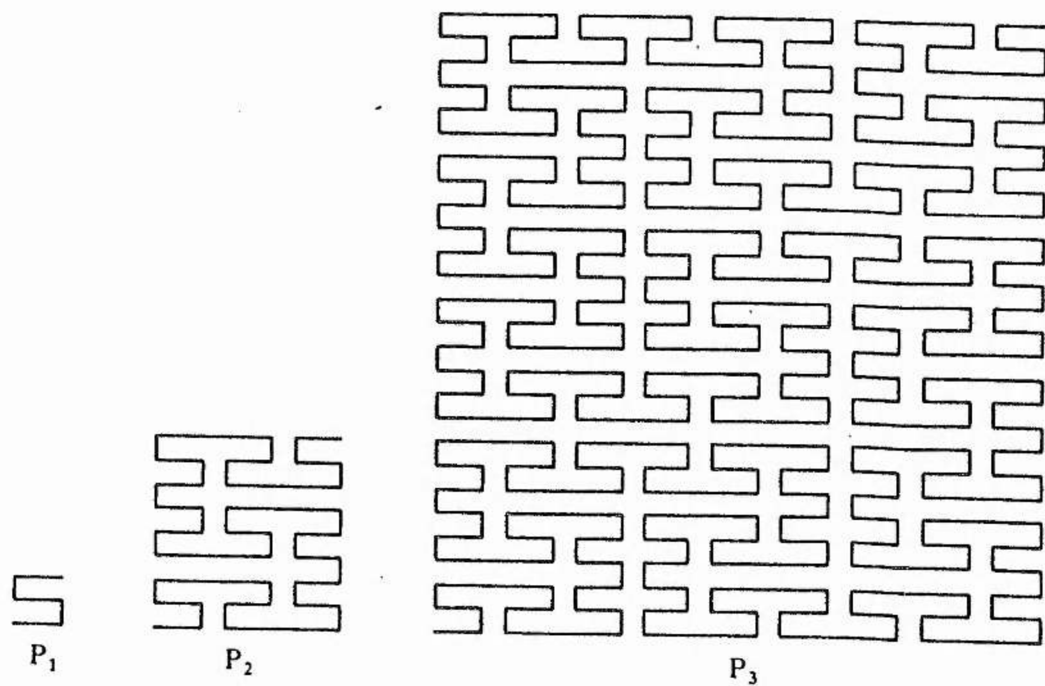


Figure 1-2.4. Peano polygons.

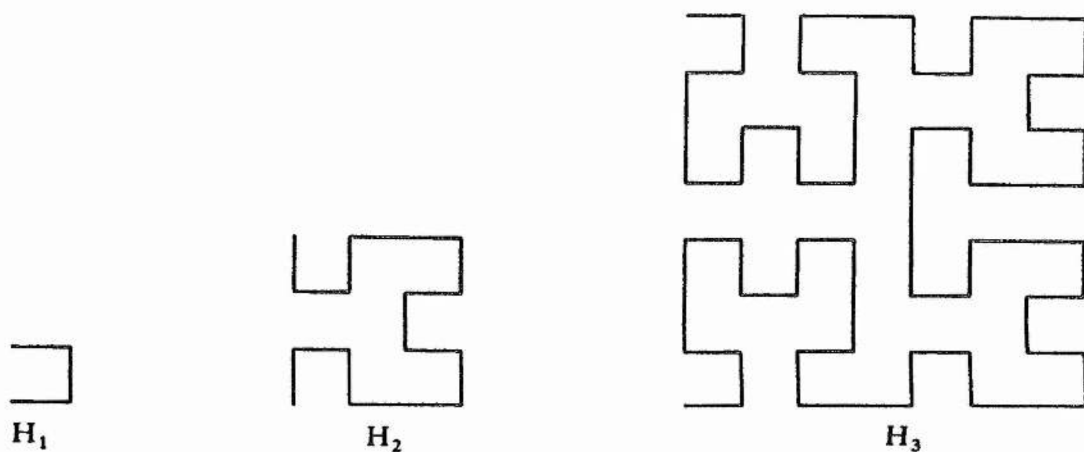
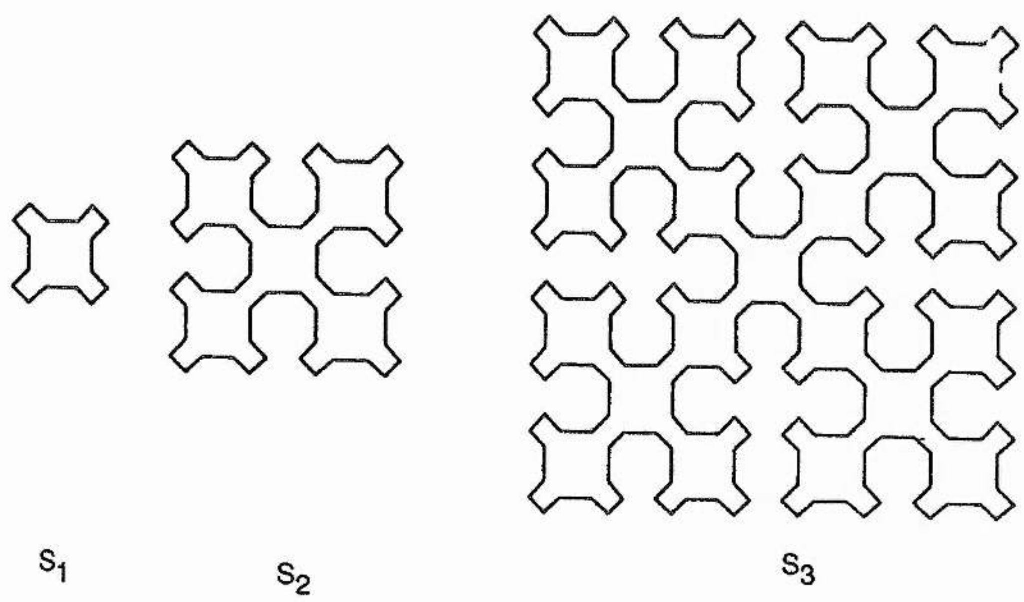


Figure 1-2.5. Hilbert polygons.



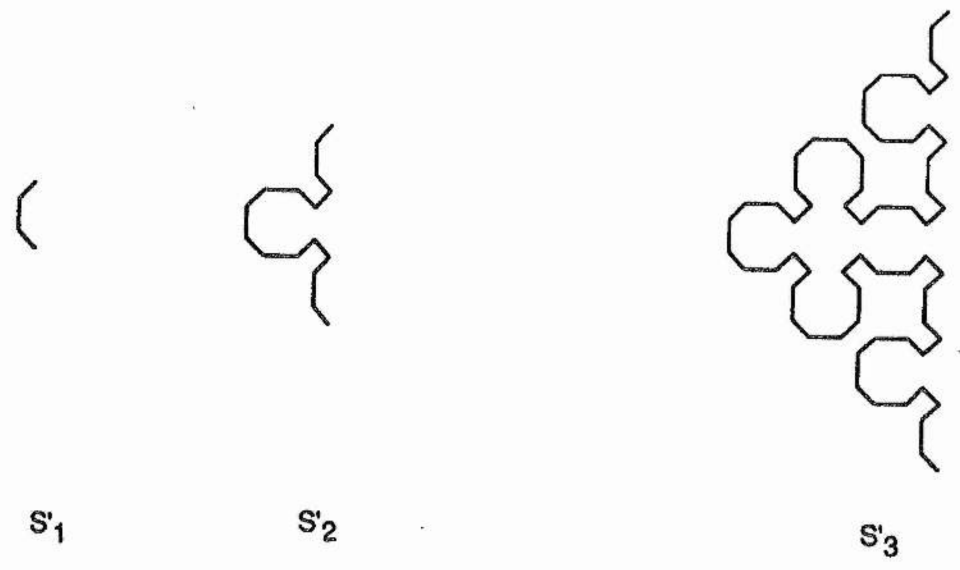


Figure 1-2.6. Sierpinski polygons(Cole[83]).

in length. Like a Peano curve its points have no unique tangent, i.e., continous but no derivative. The first four orders of Helge Von Koch's snow flake are illustrated in Figure 1-2.7.

Griffith(1986) investigated space filling curves and described a method for generating new ones. He considers the space filling curve in the unit square defined as the limit of a sequence $s_1,s_2,....$ of continuous curves which pass through every point of the square. This can be viewed as a tessellation of square tiles all of which have the same pattern but with the orientation of the pattern varying. Firstly a tile has an n x n grid marked on it and the centres of each grid-square are taken as permissible points for the construction of a continuous open path that does not intersect. The resulting path must have endpoints such that n^2 tiles can be fitted together and the individual paths joined up with standard steps as shown in Figure 1-2.8. Griffiths at this stage had shown how to generate new space filling curves which would traverse squares.

Cole(1983) has shown how Peano, Hilbert and Sierpinski polygons can all be obtained recursively from a single point. Cole showed explicit mappings between the first n non-negative integers and the n sequentially traversed vertices of any of the Peano polygons and also a generalisation of these polygons. Such polygons have been called murray polygons since they are derived using multiple radix or murray arithmetic. A formal definition and more detailed discussion of murray polygons will be given in the following chapters.

The advantages of using space filling polygons for this purpose arise from the fact that in general the curve passes through a lot of points local to

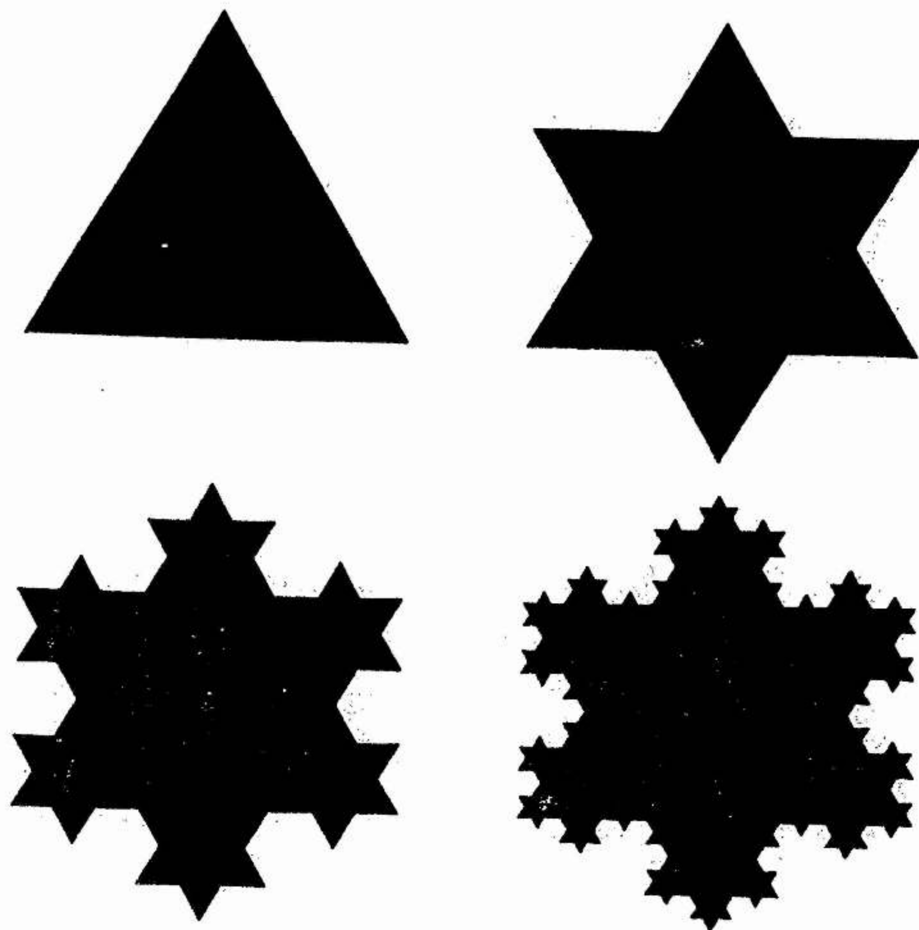


Figure 1-2.7. The first four orders of Helge von Koch's snowflake (Martin[67]).

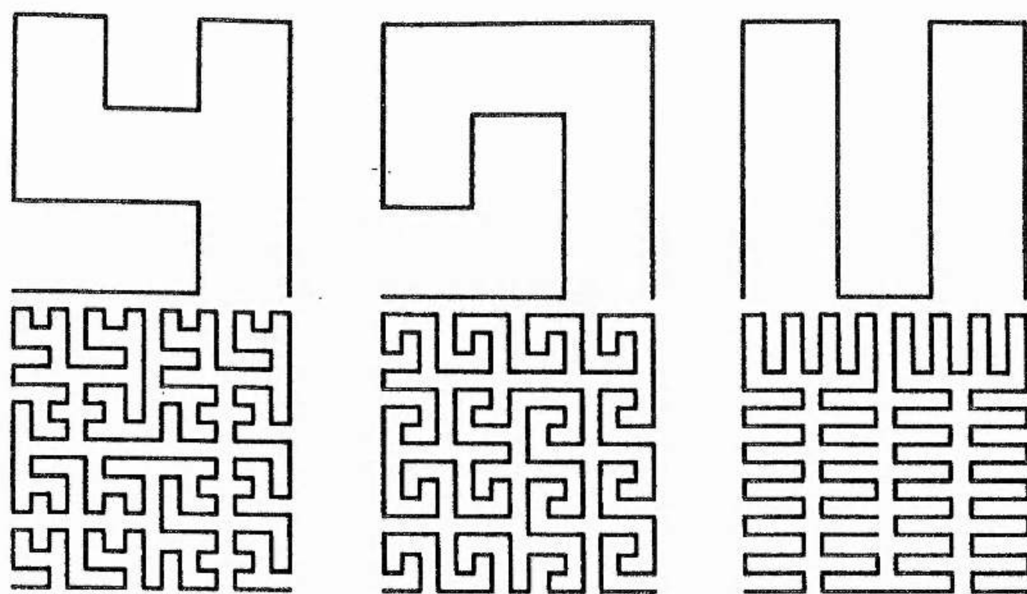


Figure 1-2.8. Three examples of basic tiles and second order polygons due to Griffiths[85])

each other in two dimensions. This pixel coherence can be exploited in many areas of image processing. This we will see in the following chapters. Moreover the Hilbert polygons include quadtree scanning as a particular, case with no additional computation or complex data structures required to record or to scan them.

1-3 Quadtree Encoding :

The term quadtree is used to describe a class of hierarchical data structures whose common property is that they are based on the principle of recursive decomposition of space. They can be differentiated on the following bases :

1. the type of data that they are used to represent,

2. the principle guiding the decomposition process,

3. the resolution (variable or not).

Quadtree representation can be used for point data, regions, curves, surfaces, and volumes.The decomposition may be into equal parts on each level (i.e., regular polygons and termed a *regular decomposition*), or it may be governed by the input. The resolution of the decomposition (i.e. the number of times that the decomposition process is applied) may be fixed, or it may be governed by properties of the input data.

Quadtrees are generated by successively dividing a two dimensional region into quadrants. Each node in the quadtree has four data elements, one for each of the quadrants in the region as illustrated in Figure 1-3.1. If all pixels within a quadrant have the same color (a homogeneous quadrant), the corresponding data elements in the node store that color. In addition, a flag is

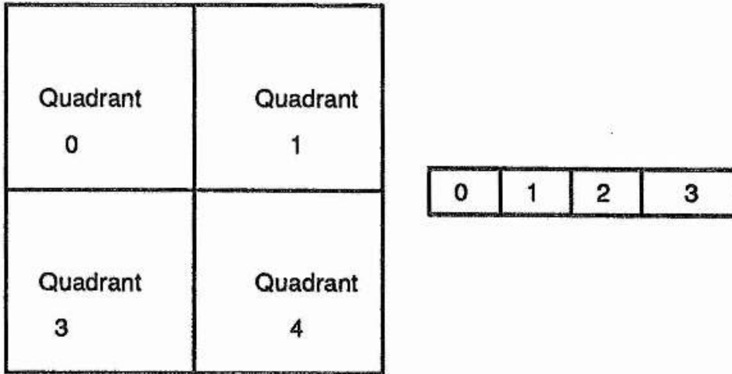


Figure 1-3.1
Region of a two dimensional space divided into numbered quadrants and the associated quadtree node with four data elements.

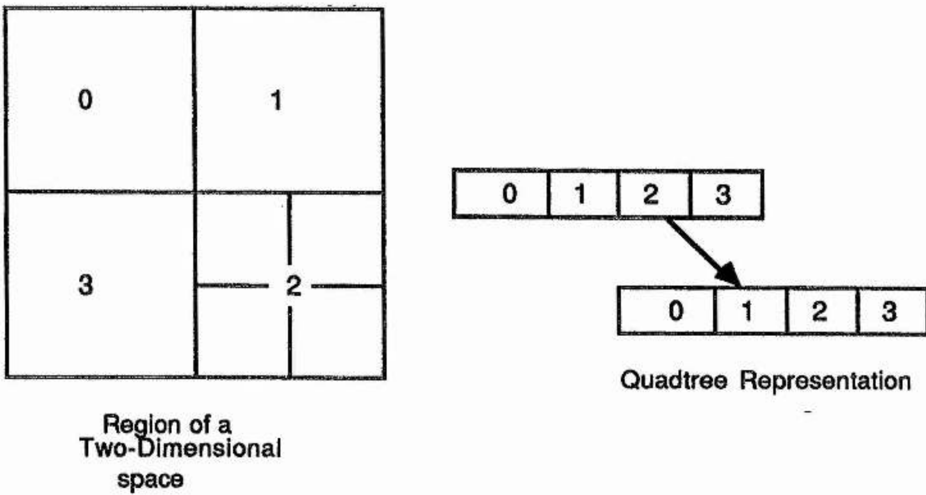


Figure 1-3.2
Region of a two-dimensional space with two levels of quadrant division and the associated quadtree representation.

set in the data element to indicate that the quadrant is homgeneous. Suppose all pixels in quadrant 2 of Figure 1-3.1 are found to be red. The color code for red is then placed in data element 2 of the node. Otherwise the quadrant is said to be heterogenous, and that quadrant is itself divided into quadrants (see Figure 1-3.2). The corresponding data element in the node now flags the quadrant as heterogeneous and stores the pointer to the next node in the quadtree. For a heterogeneous region of space, the successive subdivisions into quadrants continues until all quadrants are homogeneous. Figure 1-3.3 shows a quadtree representation for a region containing one area with a solid color that is different from the uniform color specified for all other areas in the region.

Finkel and Bentley(1974) proposed another definition for a quadtree. Here space is partitioned into rectangular quadrants. It is primarily used to represent multidimensional point data and can be referred as a *point quadtree.* In two dimensions each data point is a node in a tree having four sons. These four sons corresponds to a quadrant labeled in order NW, NE, SW, and SE. The desired record is searched on the basis of its x and y coordinates. At each node of the tree a four way comparison operation is performed and the appropriate subtree is chosen for the next test. Reaching the bottom of the tree without finding the record means that the record which we are looking at is not present in the quadtree and it can now be inserted at this position. The shape of the resulting tree depends on the order in which records are inserted into it. A *point quadtree* is illustrated in Figure 1-3.4.

Point quadtree are useful in applications that involve search. They can also be used to solve a measure problem like determination of all records within a specified distance of a given record. Search operations using point quadtrees are given in details by Bentley and Stanat(1975). Point quadtrees

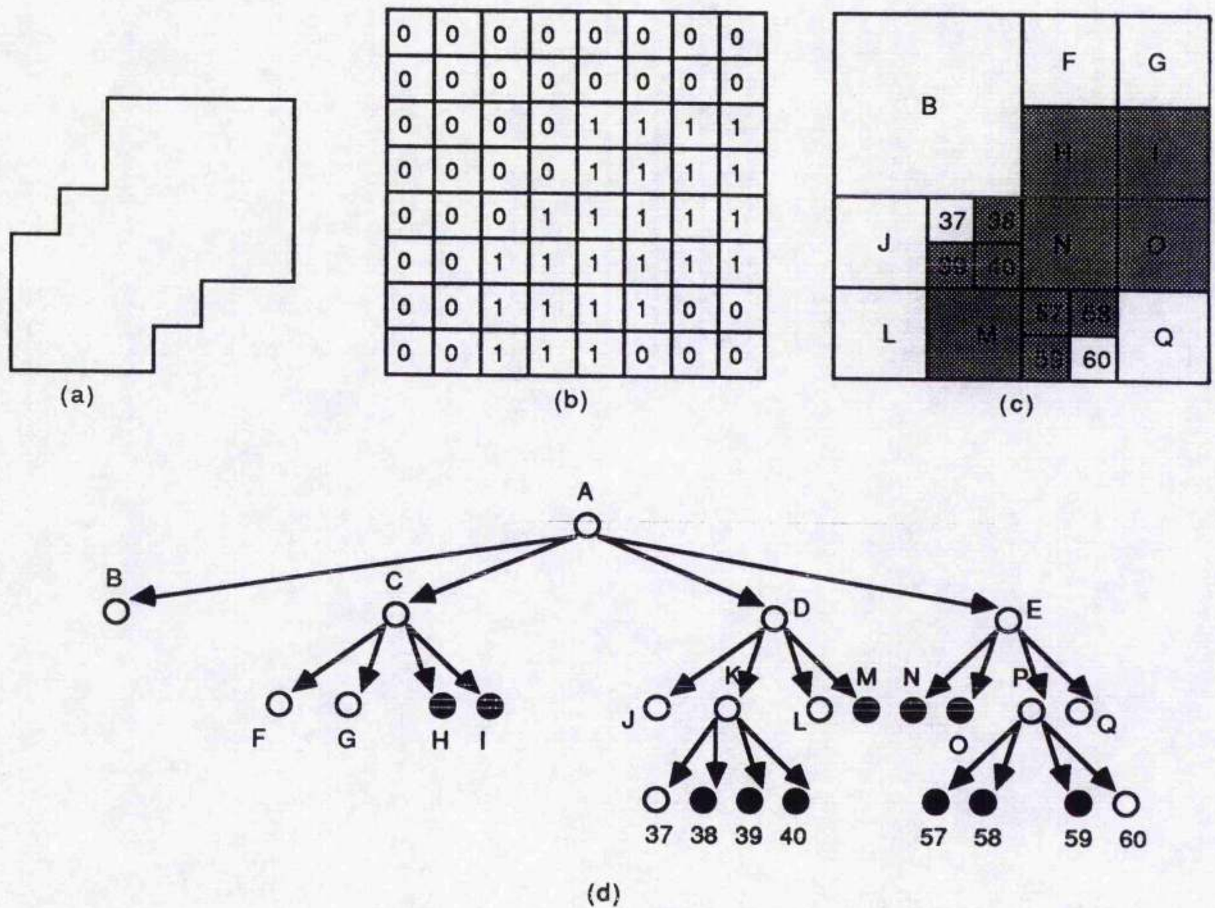
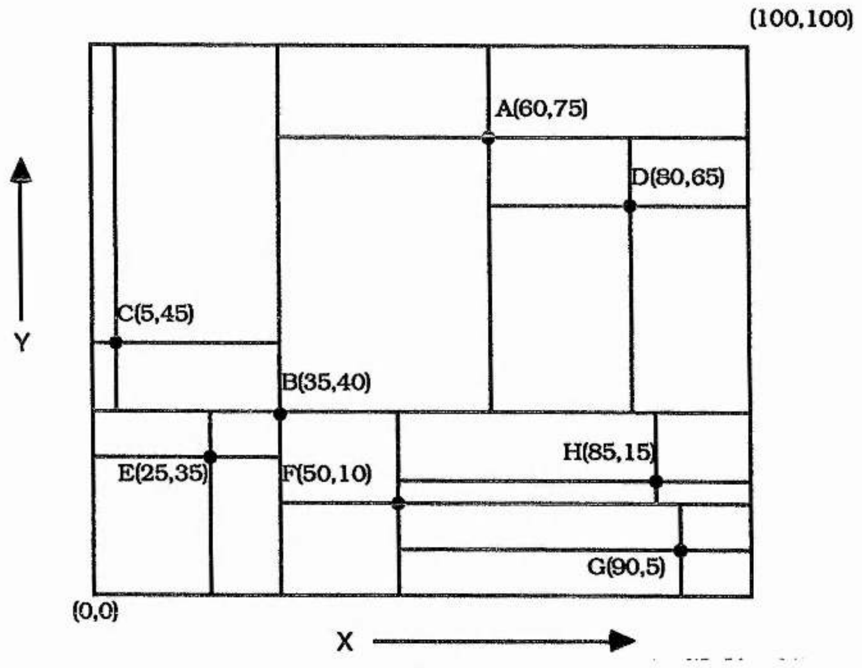


Figure1-3.3. A region, its binary array, its maximal blocks, and the corresponding quadtree.
(a) Region.
(b) Binary array.
(c) Block decomposition of the region in (a). Blocks in the region are shaded.
(d) Quadtree representation of the blocks in (c).

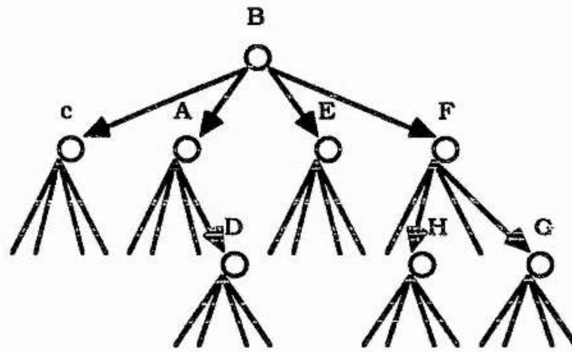
are useful with two dimensional space. As cited by Samet(1984), the problem with a large number of dimensions is that the branching factor becomes very large (i.e., 2^k for k dimensions). The storage for each node as well as for many NIL pointers for terminal nodes increases. Bentley(1975) proposed *k-d* tree, which is an improvement on the point quadtree. It avoids the large branching factors. It is a binary search tree with the distinction that at each level of the tree a different coordinate is tested when determining the direction in which a branch is to be made. In the case of two dimensions (i.e., a 2-d tree), the x-coordinates will be compared at the root and at even levels, whereas the y -coordinates are compared at odd levels. The root is assumed to be at level zero. Each node has two sons. A *k-d* tree corresponding to the point tree of Figure 1-3.4 is given in Figure 1-3.5.

An alternative tree structure that uses an analogy to the *k-d* tree given by Bentely(1975) is the *bintree* proposed by Samet and Tamminen(1984). Here, the space is always subdived into two equal-sized parts alternating between the x and the y axes. The advantage is that a node requires space only for pointers to its two sons instead of four sons. In addition, its use generally leads to fewer leaf nodes. While dealing with higher dimensional data (e.g., three dimensions) less space is wasted on NIL pointers for terminal nodes. A bintree is illustrated in Figure 1-3.6.

The problem with the tree representation of a quadtree is that it has a cosiderable amount of overhead associated with it. Moreover each node requires additional space for the pointer to its sons. This is a problem with large images that cannot fit into core memory. Consequently, there has been a considerable amount of interest in pointerless quadtree representations. They



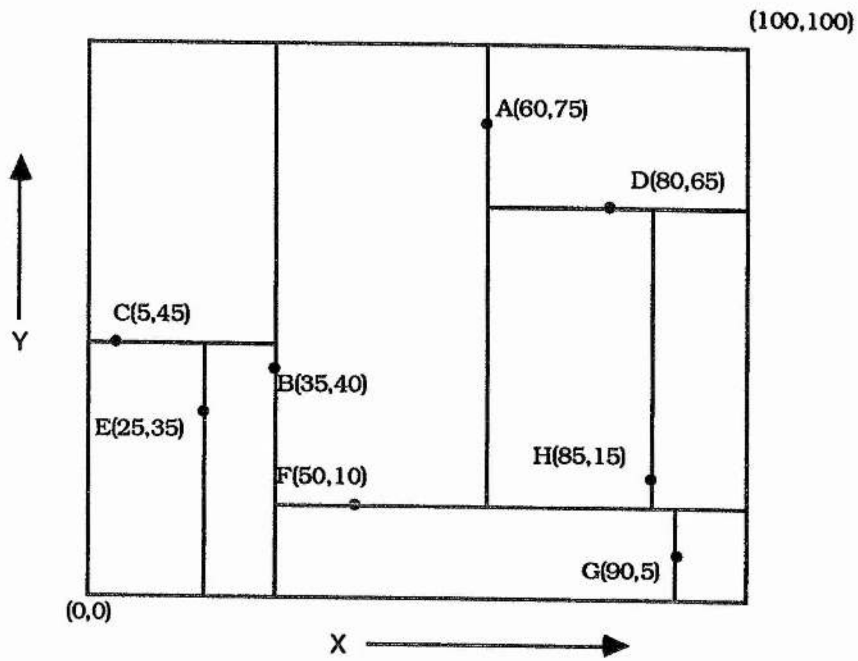
(a)



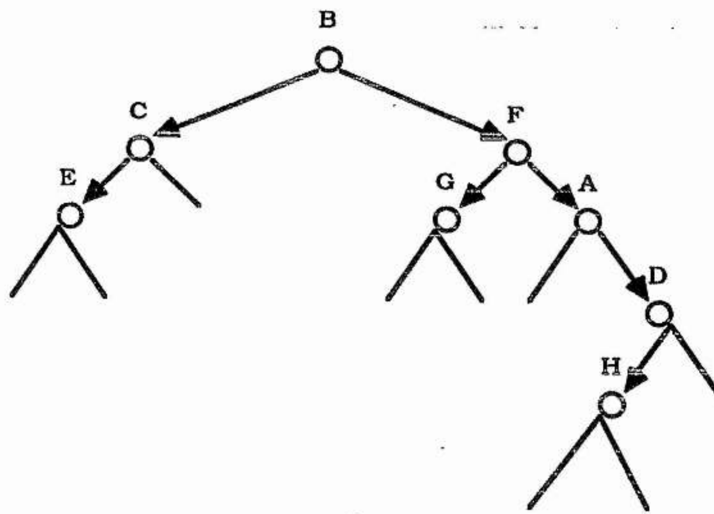
(b)

Figure 1-3.4

A point quadtree (b) and the records it represents (a).

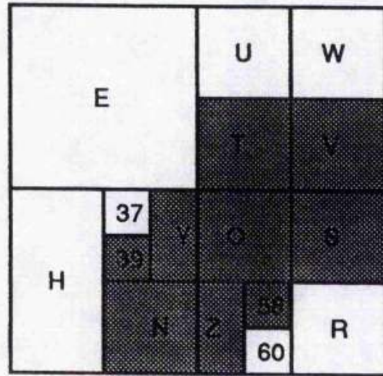


(a)



(b)

Figure 1-3.5.
A k-d tree (b) and the records it represents (a).



(a)

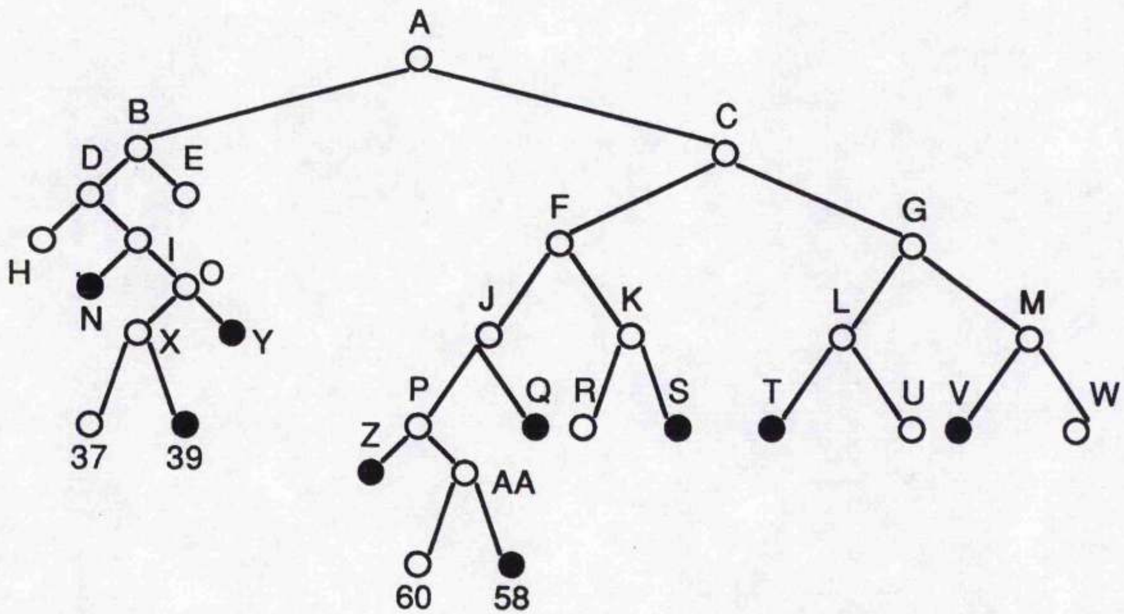


Figure 1-3.6.
The bintree corresponding to fig 1-3.3 (a) Block decomposition.
(b) Bintree representation of the blocks in (a).

can be grouped into two categories.

i. Collection of leaf nodes.

ii Traversal of the nodes of its quadtree.

1-3.1 Leaf node :

In the *leaf node* category, each leaf or pixel is encoded in a weighted quatenary code, i.e., with digit 0, 1, 2, 3 in base 4, where each successive digit represents the quadrant subdivision from which it originates. The NW quadrant is encoded with 0, the NE quadrant with 1, the SW with 2, and the SE with 3. For example, if a pixel or leaf is encoded as 321, this means that pixel or leaf belongs to the SE quadrant in the first subdivision, to the SW quadrant in the second and the NE in the third (final) subdivision (see Figure 1-3.7).

While encoding an image as a collection of *leaf nodes,* there is no need to include the locational code for every leaf node. Gargantini (1982) only retains the locational codes of the BLACK nodes and terms the resulting representation a *linear quadtree.* The codes for WHITE blocks can be obtained by using the ordering imposed by the sort without reconstructing the quadtree. All arithmetic operations on the locational code are performed by using base 4 numbers as explained above. An additional code, as a don't care, is used by Gargantini(1982), Klinger and Dyer(1976), Abel and Smith (1983), Oliver and Wiseman(1983) to yield an encoding where each leaf in a 2^n by 2^n image is n digits long. A leaf corresponding to a 2^k by 2^k block (k<n) will have n - k don't care digits. Once all the black pixels are encoded into their corresponding quaternary codes, then they are sorted and stored in an array or list. If four pixels have the same representation except for the last digit, they are eliminated from the list and are replaced with a code of (n-1) quaternary

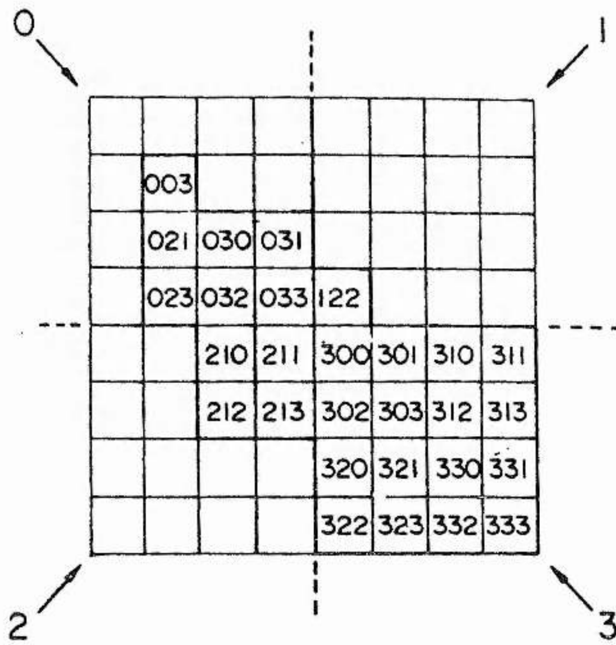


Figure 1-3.7. Quadrant labeling and generation of quaternary codes(Gargantini(1982)).

digits followed by some kind of marker (don't care), here denoted by X. For instance, if pixels 310, 311, 312, and 313 are all in the array, they can be replaced by 31X. Similarly, 30X, 31X, 32X, 33X can be replaced by 3XX and so forth, where X is the don't care digit greater than 3. Such an encoding has the interesting property that when the codes of the nodes are sorted in increasing order, the resulting sequence is the postorder traversal of the blocks of the quadtree. The main advantages of linear quadtrees, with respect to quadtrees are:

i Space and time complexity depend only on the number of black nodes.

ii. Pointers are eliminated.

Jones and Iyenger(1984) and Raman and Iyenger(1983) introduced the concept of a forest of quadtrees that is a decomposition of a quadtree into a collection of subquadtrees, each of which corresponds to the maximal square. The maximal square is identified by refining the concept of a nonterminal node to indicate some information about its subtrees. An internal node is said to be of type GB if at least two of its sons are BLACK otherwise the node is said to be of type GW.For example, in Figure 1-3.8 ,nodes C, E, and F are of type GB and nodes A, B, and D are of type GW. Each BLACK node with a label of GB is said to be a maximal square. A forest is the set of maximal squares that are not contained in other maximal squares and that span the BLACK area of the image. The forest corresponding to Figure 1-3.8 is { C,E,F}. The elements of the forest are identified by base 4 locational codes. For the path code or locational code the scheme is the same as defined by Gargantini. This type of representation can save space since WHITE items are ignored.

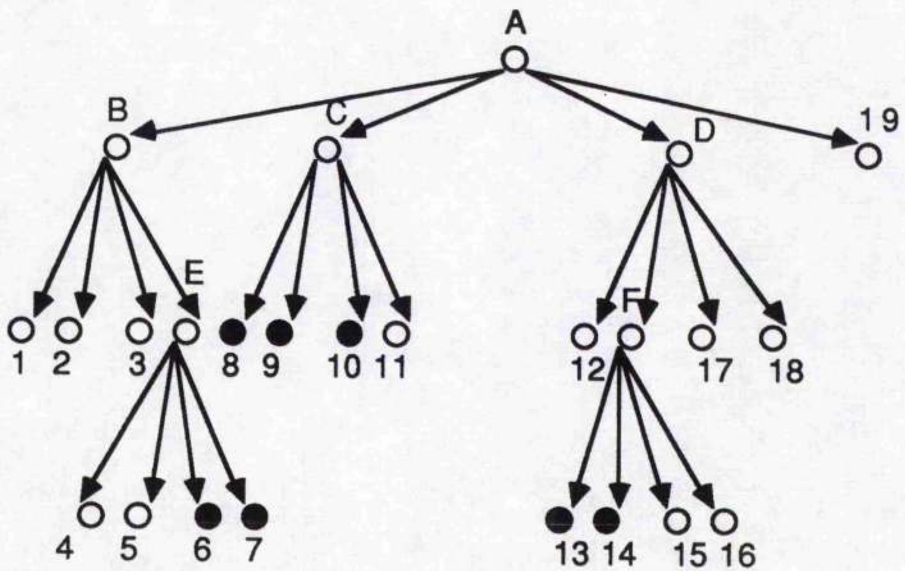


Figure 1-3.8.
A sample image and its quadtree illustrating the concept of a forest.

A linear hierarchical quadtree (LHQT) (Unnikrishnan, Venkatesh, And Shankar(1987)) is a modified version of a linear quadtree[Gargantini(1982)]. Since the level of the hierarchy indicates the size of the black nodes, the additional code, don't care which was used by Gargantini can be deleted. As a consequence, the quadtree code at level k will contain (n-k) digits only. Unnikrishnan called these modified codes the Linear Hierarchical Q-codes (LHQC). The set of all the n arrays of LHQC is called the Linear Hierarchical Quadtree(LHQT). For example, if a leaf has code 3XX, where X >= 3 , then all the additional digits i.e., X can be replaced to give a new code which will now be equal to 3 (see Table 1-1).

Level	Hierarchically ordered q-code	Linear hierarchical q-codes (LHQC)
2	244	2
1	124	12
	134	13
0	300	300
	301	301
	302	302
	320	320
	322	322

Table 1-1. The LHQT for an arbitrary binary image, (4 is the additional digit representing X).

Anedda and Felician(1988) suggested a new compression technique, referred to as *P*-compression. Here a pixel code be divided into a prefix of P digits, 1 <= P < n, and a suffix of (n-P) digits. Every pixel belonging to the quadrant originated by the first P quadrant subdivisions that is consequently of size $2^{(n-P)}$ pixels has the same prefix. Then store all distinct prefixes once; each of them will be followed by the number of pixels having that prefix,

and by the corresponding suffixes (see Figure 1-3.9a). They have compared the results with those obtained by Gargantini(Linear-quadtree). Two cases namely the best case and the worst case, are considered (see Figure 1-3.9 a,b, and c). In the best case Gargantini's compression algorithm is shown to be more efficient than *P*-compression, since the quadtree compression consists of a single pixel code with m don't care digits(where m is the size of a quadrant), in its rightmost positions, whereas a *P*-compressed quadtree needs more codes, that is more storage space.It has been pointed out that in the worst case *P*-compression is better if m<=4.

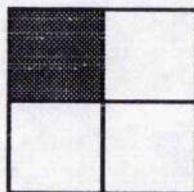
1-3.2 Traversal of the nodes of its quadtree :

The second pointerless representation is in the form of a preorder tree traversal (i.e., depth first) of the nodes of the quadtree. The result is a string consisting of the symbol "(" , "B", "W" corresponding to GRAY (i.e., if all pixels within a quadrant are not of same color), BLACK, and WHITE nodes respectively. This representation is due to Kawaguchi and Endo (1980) and is called *DF-expression.* For Example, the image of Figure 1-3.3 has (W(WWBB(W(WBBBWB(BB(BB(BBBWW as its DF-expression (assuming that sons are traversed in the order NW, NE, SW,SE). The original image can be reconstructed from the *DF-expression* by observing that the degree of each nonterminal i.e., GRAY node is always 4.

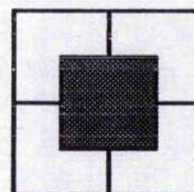
Oliver and Wiseman(1983) also reported a linear code which specifies a quadtree in depth first order. Their data items consist of five-bit numbers, of which the last four bits constitute the color value. If leaf, color the square with value in 4 bit field, otherwise the color value refers to an average of the quads beneath. Figure 1-3.11(a) illustrates a simple quadtree in their encoding. In the above coding, the value which indicates a non-leaf quad is

Linear Quadtree	P-Compressed Quadtree
003230020 003230022 003230023 003230031 003230031 003230033	0032300 5 20 22 23 31 33

(a)



(b)



(c)

Figure1-3.9.

(a) P-compression coding to a linear quadtree

(b) Best case position of a 2^m by 2^m region in a 2^n by 2^n binary image.

(c) Worst case position of a 2^m by 2^m region.

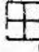
repeated many times. The compressed quadtree scheme was designed by Woodwark(1984) in order to obtain the compression of the node list resulting from a depth-first visit of the quadtree. This method consists in associating a type code with each non-terminal node in the tree. There are 52 possible type codes for the sons of any non-leaf quadrant (see Figure 1-3.10). These allow for any or all of the sons of that quadrant to be defined further down the tree. If, for example a non-leaf quadrant has dependent leaf of two colors, A and B, the record of representing that quadrant will consist of the appropriate type code, followed by the color values of A and B in the order defined by the type code. Any dependent non-leaf quadrants will follow in traversal order. Figure 1-3.11(b) shows a simple quadtree represented using this compressed traversal code.

The quadtree is proposed as a representation for binary images because its hierarchical nature facilitates the performance of a large number of operations. Most images are traditionally represented by structures such as binary arrays, raster(i.e, run length), chain code(i.e., boundaries) or polygons(vectors), some of which are chosen for hardware reasons (e.g., run lengths are particularly useful for rasterlike devices such as television) . Conversion from these methods to quadtrees is given in Samet(1984,1981b), Unnikrishnan, and Venkatesh(1984).

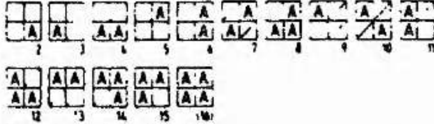
1-4 Volume Data

Extension of the quadtree to represent three-dimensional objects by use of octrees has been proposed independently by many researchers Hunter(1978); Jaclins and Tanimoto(1980); Meagher(1982); Reddy and Rubin(1978) as cited by Samet(1984). The process begins with a 2n by 2n by 2n object array of unit cubes or voxels(volume elements)

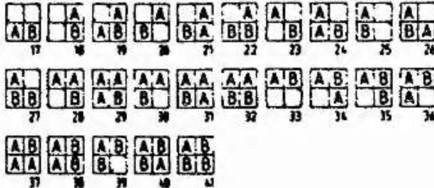
No leaf quads



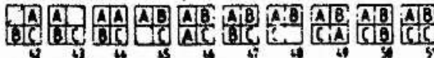
One colour of leaf quad



Two colours of leaf quad



Three colours of leaf quad

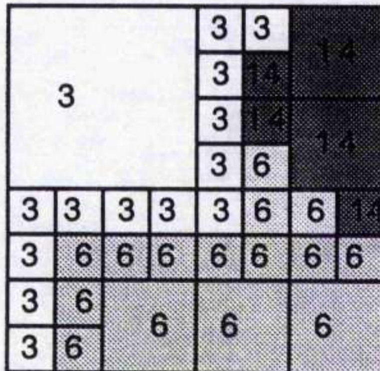


Four colours of leaf quad

A B C D indicate leaf quads whose colours follow the type code

indicates non-leaf quads whose own type codes follow later in the traversal

Figure 1-3.10. Type code for compressed traversal coding (Woodwark(1984)).



(a) Oliver and Wiseman's treecode

22 20 20 3 3 6 6 20 3 3 6 3 6 20

6 3 6 3 3 22 6 21 6 3 6 6 6 24 6

6 6 14 26 22 3 3 6 14 22 3 3 14 3 14 14

Underlined numbers are average values (+16 for identification) at non-leaf quads.

(b) Compressed traversal code

9 3 6 14 29 3 14 46 3 14 46 3 14 6

2 6 29 3 6 32 3 6 38 3 6 4 6 41

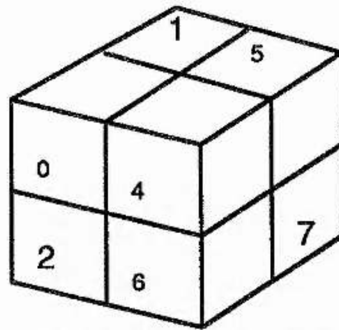
3 6 37 6 14

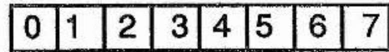
Underlined numbers are type codes (see Figure 1-3.10)

Figure 1-3.11.
Traversal and compressed traversal coding of a simple quadtree (Woodwark[1984]).

[Jaclins and Tanimoto(1980)] (also termed as obels [Meagher(1982)]. The octree is an approach to object representation similar to the quadtree, and is based on the successive subdivision of an object array into octants. If the array does not consist entirely of 1's or entirely of 0's, then it is subdivided into octants, suboctants, etc. until cubes (possibly single voxels) are obtained that consist of 1's or 0's; that is they are entirely contained in the region or entirely disjoint from it. This process is represented by a tree of out degree 8 in which the root node represents the entire object with octants labelled as in Figure 1-4.1, and the leaf nodes are said to be BLACK or WHITE , depending on whether their corresponding cubes are entirely within or outside of the object, respectively. All nonleaf nodes are said to be GRAY. Figure 1-4.2 contains an example object in the form of a staircase and its corresponding octree. The labels denote the octant numbers associated with each son by using the labelling convention of Figure 1-4.2.

Many of the algorithms obtained for the quadtree, can be extended to the octree. Gargantini(1983) makes use of a pointerless representation termed a linear octree (analogous to the linear quadtree, Gargantini(1982). He represented each pixel by an octal integer in a weighted system. Thus the digits of weight 8^{n-h}, $1 <= h <= n$ identifies the largest octant to which the pixels belong at the hth subdivision. In the planar case, a quadrant is subdivided into four squares identified by NW,NE,SW and SE. An additional notation "Forward" and "Backward" (F and B) has been introduced to distinguish between the four cubes nearer to the viewer with respect to the other four cubes. Here octant NWF is encoded with 0, octant NEF with 1, octant SWF with 2, octant SEF with 3, octant NWB with 4, octant NEB with 5, octant SWB with 6, and in the last octant SEB with 7.





Data Elements in the
Representative Octree Node

Region of a Three-Dimensional Space

Figure 1-4.1.
Region of a three-dimensional space divided into numbered octants and the associated octree node with eight data elements (octant 3 is not visible) .

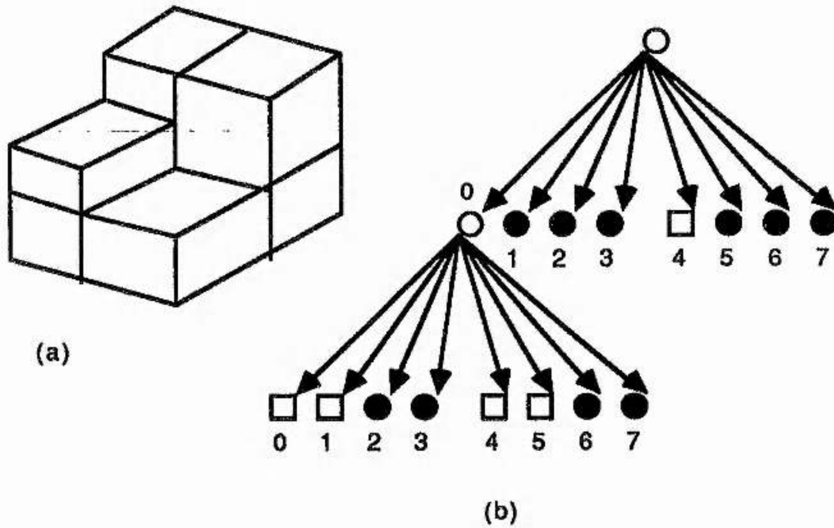


Figure 1-4.2.
Example object (a) and its octree (b).

● = BLACK = "Full";

□ = WHITE = "VOID" (empty);

○= GRAY (BLACK and WHITE).

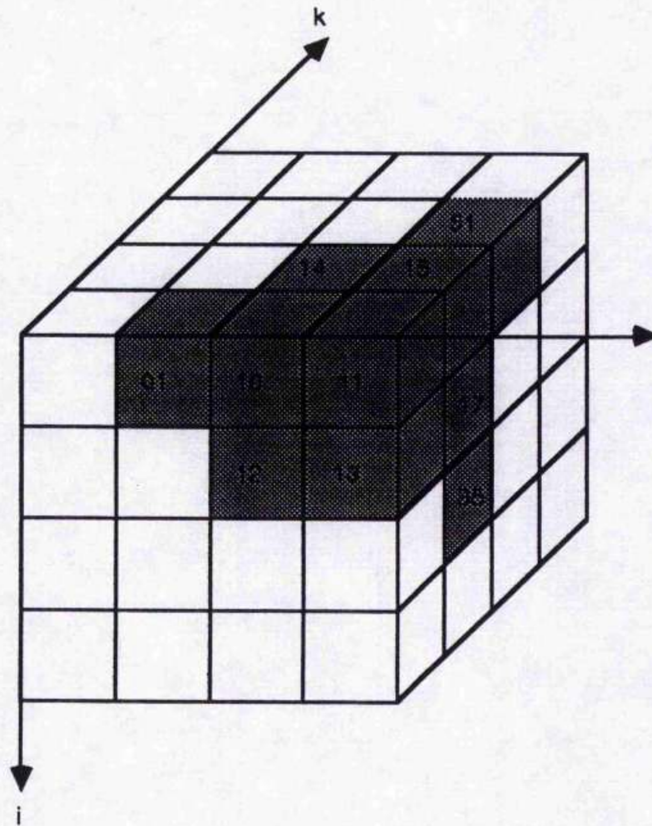


Figure 1-4.3
Representation of an object with n=2.

Once all the black pixels are encoded, condensation can be applied as discussed in Gargantini(1982) i.e., the representation for the region shaded in Figure 1-4.3(Gargantini(1983).

{ 01,10,11,12,13,14,15,16,17,35,51}

will be replaced by,

{01,1X,35,51}

where X is the marker or don't care digit defined earlier, with an integer>7.

All the algorithms, explained above for octrees, have a common disadvantage. The entire three-dimensional image array must be loaded into computer memory from the start and left there throughout a session.

Yau and Srihari(1983), proposed a general approach to construct a 2^d-tree (or hyperoctree), representing a d-dimensional binary image from the 2^{d-1}-trees representing (d-1)-dimensional cross sections (or slices) of the image, orthogonal to any of the axes. The word hyperoctree is defined by Yau and Srihari, for a d-dimensional binary image. Here a d-dimensional image is recursively divided into 2^d hyperoctants giving a 2^d-tree or hyperoctree. Since the work given in this thesis is based on two-dimensional and three-dimensional images, we need not discuss d-dimensional images in this report. The quadtree to octree conversion algorithm developed by Yau and Srihari is established in such a way that 2^n quadtrees $q_0, q_1, \ldots q_{2^n-1}$, all of which are generated from the array of side 2^n, are sequentially loaded. Then q_1 is merged with q_0, q_3 is merged with q_2, ..., q_{2^n-1} with q_{2^n-2}, to give 2^{n-1} new trees $q'_0, q'_1, \ldots q'_{2^n-2}$. Repeating such merging steps n times, we obtain the octree. The only operation at every merging is to copy the subtree whose root is at a certain depth onto the corresponding node of the other tree while traversing the two trees in parallel, this is explained in Figure 1-4.4. Just to explain the theory discussed above, we consider eight quadtrees, whose origin is not known. For more details see Xiaoyang, Tosiyasu ,Fujishiro, and Noma(1987) and Chien and Aggarwal(1986), and Shrhari(1981).

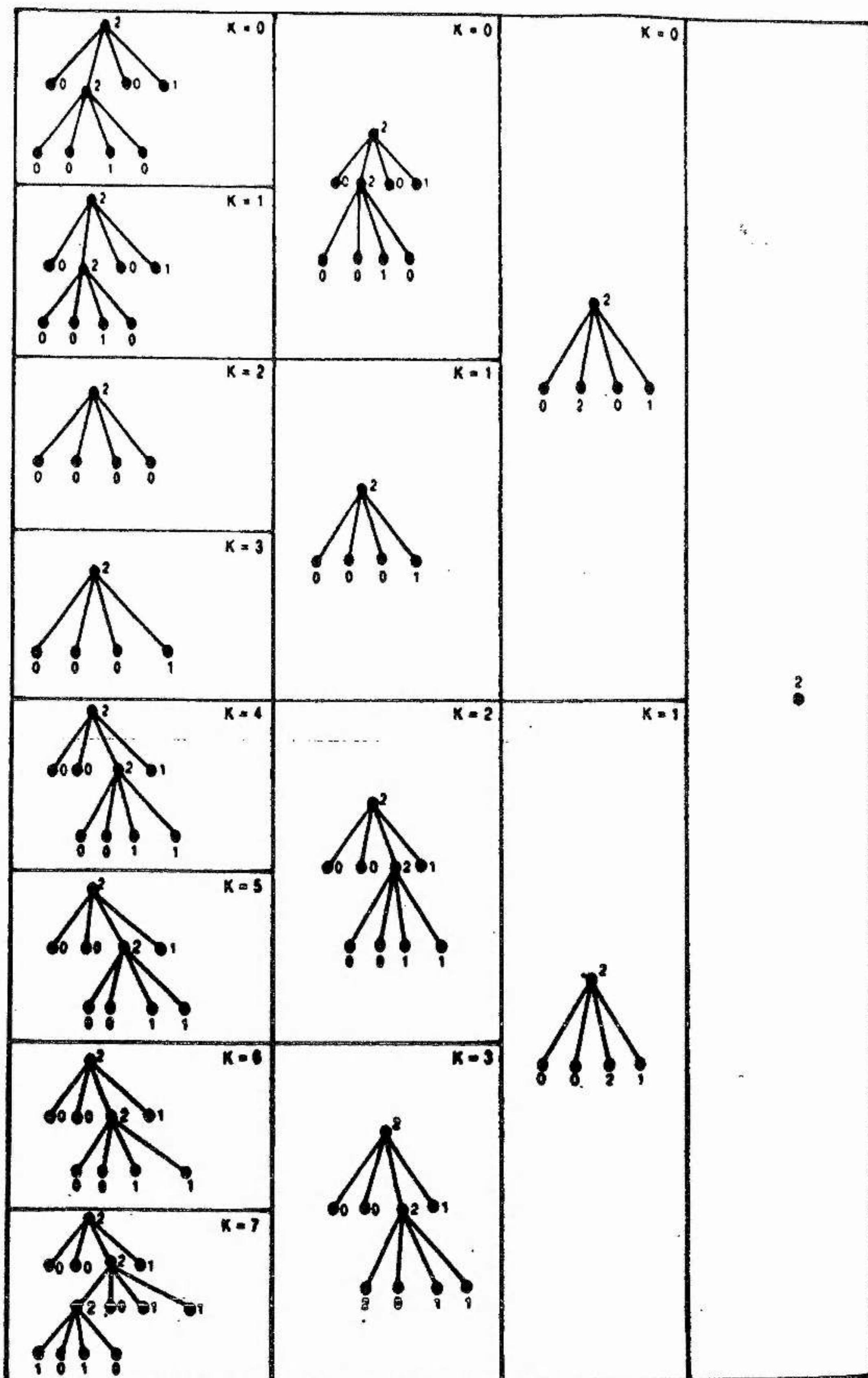


Figure 1-4.4. Generation of the octree of an object using quadtree of its slices [Yau, and Srihari(1983)]

Chapter 2

2. **MURRAY POLYGONS** 20

2 - 1 Introdution 20

2 - 2 Murray Polygons 20

Gray Codes or Cyclic Progressive Numbers 23

Direct Peano Transformations 25

Murray Arithmetic 26

Murray Transformation 28

Mixed Scan 31

Some Lemmas 32

Implementation Of Murray Scan 39

Original Implementation 39

A Faster Murray Scan Algorithm 42

Hardware Implementation 46

Three-Dimensional Cartesian Coordinates 46

Extension To 3-D And n-D Murray Polygons 47

Method One 47

Some Lemmas 50

Second Method 52

Polar Murray Scan 54

Polar Coordinates 54

Changing Coordinates Systems 55

Graphs In Polar Coordinates 56

3D And Higher Dimensional Polar Coordinates 56

Cylindrical And Spherical Coordinates 56

Implementation Of Planar Polar Murray Scan 59

Cylindrical Polar Murray Scan 61

Spherical Polar Murray Scan 61

2-2.11 Application Areas 62

Scanning 62

2-2.12 Remarks 63

2-1 Introduction:

In this chapter the developments by Cole(1985a,1985e) from Peano's original concept of a space filling curve to the definition of murray polygons and their associated methods via multiple radix arithmetic are described. Two dimensional and three dimensional murray polygons for cartesian as well as for polar coordinates are described in detail. The formal definition of a space filling curve and the literature has been discussed in detail in Chapter 1.

2.2 Murray Polygons:

Cole became interested in space filling curves motivated by an argument with a colleague over the categorization of curves as either Hilbert or Peano polygons. In fact Peano, introduced the idea of an explicit space filling curve and Hilbert and others introduced the idea of limiting sequences of polygons leading to space filling curves.

After initial investigations Cole produced neat algorithms for drawing the common space filling curves, all three (i.e., Peano, Hilbert, and Sierpinski), are obtained recursively from a single point [Cole(1983)]. The main procedure for the Peano polygon is given in Program 2.1, the language used is the Outline System of PS-algol [Carrick,Cole, and Morrison(1987), and Morrison(1988)]]. Some confusion arises from the polygon P_1 which appears to have only six vertices rather than the nine. This problem is resolved by assuming that the Peano polygon has nine vertices by splitting each edge at its mid-point. This proved to be of significance later.

```
let draw.peano  = proc ( cint complexity )
begin
   let peano = proc( cpic p ; cint complexity, order, old.width, width  );
   nullproc
   if order = complexity then draw( p, 0, width, 0, width )
   else
   peano   (        scale rotate  ( p ^
                shift scale p by 1,-1 by old.width + 2, width ^
                shift p by 2 * old.width + 4, 0 ) by -90 by -1,1,
                complexity, order + 1, width, 3 * old.width + 4
           )

        peano( [ 0,0 ],2 * complexity, 0, 0, 0)
end
```

Program 2.1. Main algorithm for Peano Polygons.

Peano's original definition had taken a point on the interval [0,1] and split it into two real base three numbers by taking all the odd indexed digits in their sequential order for the value for x and all the even ordered digits in their sequential order for the value for y to obtain the point (x,y) of the square. It should be noted that to maintain the uniqueness of the transformation, the ambiguity of real number representation was dealt with in a more complex manner. Before we talk about the ambiguity problem it will be better to discuss the transformation which is used by Peano. Let $T = 0.a_1 a_2 a_3.......$ be a sequence of digits each in base 3 representation. This sequence is now split into two real sequences,

$$X = 0.b_1 b_2 b_3.........., \qquad Y = 0.c_1 c_2 c_3..............$$

where digits b_i and c_i are given by the relation,

$$b_n = K^{a_2+a_4+\ldots\ldots+a_{2n-2}} (a_{2n-1}), \quad \text{(A)}$$

$$c_n = K^{a_1+a_3+\ldots\ldots+a_{2n-1}} (a_{2n}), \text{ for } n = 1,2,3,\ldots\ldots \quad \text{(B)}$$

where for a digit 'a'; K(a) represents the reduced radix complement of 'a' i.e., K(a) is equal to 2-a. The term $K^n(a)$ represent the operation K repeated n times on digit 'a'. where,

$K^n(a)$ = a if n is even

K(a) or 2-a otherwise.

Surprisingly, the above result is very similar to that of the Gray code transformation. One can conclude that Peano invented the Gray code transformation before Gray did. The Gray code transformation is discussed in the next section.

From the above results we can say that the digit b_n i.e., the nth digit of X is equal to a_{2n-1} which is the odd numbered digit or to its complement according as the sum $a_2+a_4+\ldots\ldots+a_{2n-2}$ of digits of even rank is even or odd. Similarly for the Y digits we will consider the sum of digits of odd rank.

Peano showed that if a sequence T is given, then we can determine X and Y and if X and Y are given then a sequence T can be determined. Peano's transformation gives the same values for the two sequences given as,

$$T = 0.a_1a_2a_3\ldots\ldots\ldots a_{n-1}a_n222\ldots\ldots\ldots,$$

where a_n is equal to 0 or 1, and the other,

$$T' = 0.a_1a_2a_3\ldots\ldots\ldots a_{n-1}a'_n000\ldots\ldots\ldots,$$

where $a'_n = a_n + 1$.

The value of the sequence T is given as,

$$t = \text{val } T = a_1/3 + a_2/3^2 + \ldots\ldots\ldots\ldots + a_n/3^n + \ldots\ldots\ldots$$

The correspondence between T and (X,Y) is such that if T and T' are of different form, but val T = val T', and if X, Y are the sequences corresponding to T, and X',Y' are the sequences corresponding to T' then we have,

$$\text{val } X = \text{val } X', \qquad \text{val } Y = \text{val } Y'.$$

Note that two decimal fractions of different form, such as 0.022222..... and 0.1000....... have the same value, but the correspondence between the two different forms and the associated numbers X and Y do not have identical representations. Actually if we split the above two numbers using the Peano transformation then the two pairs of numbers will be (0.022.....,0.222....) and (0.100......, 0.222.....) , which are same in value, but have different representations.

Cole considered the application of a similar technique to define a mapping from the first 3^{2n} base three integers to the vertices of the nth Peano polygon. He tried various alternative transformation but they all failed, until the idea of using Gray códes, or cyclic progressive numbers occured.

2.2.1 Gray Codes or Cyclic Progressive Numbers :

Gray(1953) discussed ways in which cyclic progressive number systems could be defined and Cole(1966) gave conversion rules and addition and multiplication tables for such systems. Cyclic progressive integers have the property that successive integers differ in only one digit. They are not restricted to binary representation but can take any number base. The following conversion rules from a pure number to a Gray code is given by Cole. Two cases should be considered; 1) odd base and 2) even base systems.

Suppose

$$d = d_n d_{n-1} \ldots\ldots\ldots d_2 d_1.$$

is an integer in a pure number system with radix r. Then the Gray code transformation d' of d is defined as,

$$d' = d'_n d'_{n-1} \ldots\ldots\ldots d'_2 d'_1.$$

The value for the digits d'_i depends upon the radix r.

Case 1. r is odd.

$d'_i = d_i$ if the sum of all its more significant (i.e. left hand) digits is even,

$= r - 1 - d_i$ otherwise.

Case 2. r is even.

$d'_i = d_i$ if d_{i+1} is even,

$= r - 1 - d_i$ otherwise.

Note: The term $r - 1 - d_i$ is the usual ***reduced radix complement.***

In both cases, the conversion rule back to ordinary integer(pure) number form is exactly the same as given in the odd case above. Table 2.1 gives some simple example with different bases.

base 3	
Pure	Gray code.
0000	0000
0001	0001
0002	0002
0010	0012
0011	0011
0012	0010
0020	0020
0021	0021
0022	0022
0100	0122

base 4	
Pure	Gray code.
0000	0000
0001	0001
....	....
....	....
0010	0013
0011	0012
0012	0011
....	....
0333	0303
1000	1333

Table 2.1. Conversion from pure integers to Gray code integers with different bases.

2.2.2 Direct Peano Transformations :

Cyclic progressive number system and space filling curves have a similarity. The successive cyclic progressive integers differ in only one digit, whereas in the case of Peano, Hilbert and other space filling curves, the consecutive vertices are only one unit apart in either x or y but not both. After considering several possibilities for base two numbers Cole found the transformation for the case of Peano polygons by using base three numbers (further details follow in section 2.2.3.2). Further he realised the importance of the commutability of conversion to Gray codes and reduced radix complementation to the mapping, that is if,

a' is the gray code equivalent of a, and a^* is the reduced radix complement then

$$(a')^* = (a^*)' \qquad (1)$$

This result is only true for odd base numbers. For Hilbert polygons this result is not true, since the obvious corresponding base is even. The proofs and detailed explanations to cover explicit mappings from the first n^{2p} base

Gray code integers into the ordered vertices of the pth Peano polygon and vice versa are to be found in Cole (1985a). The result used applies to any odd based number system with radix r, giving a generalised Peano polygon P_m n,r of type r in n dimensions which passes through all r^{mn} points with integer coordinates in the r-dimensional cube of side length r^m-1 (m = 1,2,3 ....).

The transformation to the pth Peano polygon is only true for the first n^{2p} integers where n is odd. Soon after this Cole (1985c) produced a table driven mapping between the first 2^{2p} integers and the ordered points of the pth Hilbert polygon. Griffith(1985) also derived a table-driven algorithm for the generation of Hilbert curves. These methods with minor modifications can be made equivalent. As with the Peano transformation the method could be extended to deal with Hilbert polygons in higher dimensional space.

2-2.3 Murray arithmetic :

All the techniques including quadtree methods, which have been discussed so far are limited to squares with a restriction on their dimensions. The tool to escape from the square cell was derived by Cole(1985b) using multiple radix arithmetic, in short murray arithmetic.

Murray arithmetic is integer arithmetic in a number system in which each murray integer is defined as a sequence of digits

$$d_n, d_{n-1}, d_{n-2} \ldots\ldots, d_1$$

together with a sequence,

$$r_n, r_{n-1}, \ldots\ldots\ldots, r_1 \text{ of integers}$$

where r_i defines the radix associated with d_i (for $i = 1,2, \ldots ,n$) such that for each i we have,

$$0 \le d_i \le r_i - 1.$$

The main operation required is the addition, which is defined as usual, except that carry now takes place from the ith to the (i+1)th digit when the sum in the ith place exceeds $r_i - 1$. Addition may only take place between integers having an identical radix sequence. Further two successive murray integers will either differ in only one digit (i.e., d_1) or carry takes place from the first to the ith digit. The reduced radix complement for murray integers is defined as :

$$d^* = b = b_n b_{n-1} \ldots\ldots b_1,$$

where, $b_i = r_i - 1 - d_i \quad (i = 1,2, \ldots\ldots , n),$

and its gray code equivalent is given as :

$$d' = c = c_n c_{n-1} \ldots\ldots c_1,$$

where,

Case I. All r_i odd :

$$c_i = d_i \quad \text{if the sum of } d_n, d_{n-1}, \ldots, d_{i+1} \text{ is even or if } i = n$$

$$= r_i - 1 - d_i \quad \text{otherwise.}$$

Cole referred to these murray integer as *murray-o* integers.

Case II. All r_i even :

Here d_i is replaced by its reduced radix complement if d_{i+1} is odd or $i \sim= n$, and unchanged otherwise. These murray integers were referred to as *murray-e* integers.

Case III. r_i even or odd :

Consider any digit d_i with corresponding radix r_i. Let $j > i$ be the first integer such that r_j is even. Always assume that r_{n+1} is even.

Let

$$p_{i,j} = \left(\sum_{k=i+1}^{j} r_k \right) \text{ rem } 2.$$

Then the cyclic progressive transform d' of d is

$$d' = c_n c_{n-1} \ldots c_1,$$

where

$$c_i = d_i \quad \text{if} \quad p_{i,j} = 0$$

and

$$c_i = r_i - 1 - d_i \quad \text{if} \quad p_{i,j} = 1.$$

Cole referred to these murray integer as *hybrid murray* integers.

2-2.3.1 Murray Transformation:

Cole(1985b) proved that the *murray-o* integers can be transformed, using a similar method to that described in the section for Peano polygons, such that all the points with integer coordinates within a rectangle m by n,

consecutive points being not more than one. Importantly the murray transformation applies to each possible factorisation of p and q taken in any order. The resulting space filling curves he named murray polygons. He also extended this generalisation of the Peano polygon to higher dimensional space.

Cole realised that murray polygons are not restricted to odd dimensions as the restriction on the radices being odd can be lifted for the first and last radices giving even sided rectangles in 2-dimensions (see Figure 2.1). Considering the two-dimensional cases we now have an explicit transformation from the first m positive integers to the m points with integer coordinates in a rectangle containing exactly m such points. The stages are outlined as follows:

Express the fixed base number d as a murray integer with given murray radices, r_i say,

$$d = d_n d_{n-1} \ldots d_2 d_1 \,, \; (0 \leq d_i \leq r_i - 1,\; i = 1,2, \ldots, n (=2k, \text{ where k is an integer})).$$

Convert this murray integer to a Gray code integer

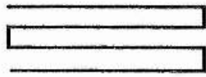
$$d' = c_n c_{n-1} \ldots c_2 c_1 .$$

Split the Gray code number into parts x' and y' as below

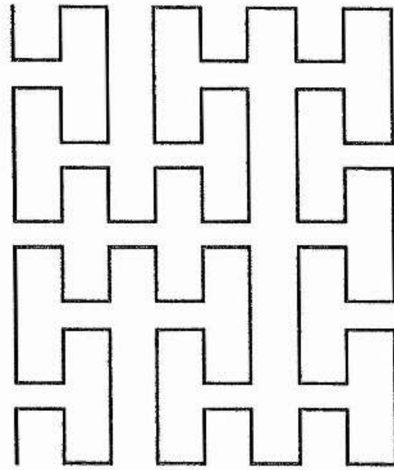
$$x' = c_{n-1} c_{n-3} \ldots c_3 c_1 \qquad y' = c_n c_{n-2} \ldots c_4 c_2 .$$

Convert Gray coded x' and y' separately back into murray integers x" and y".

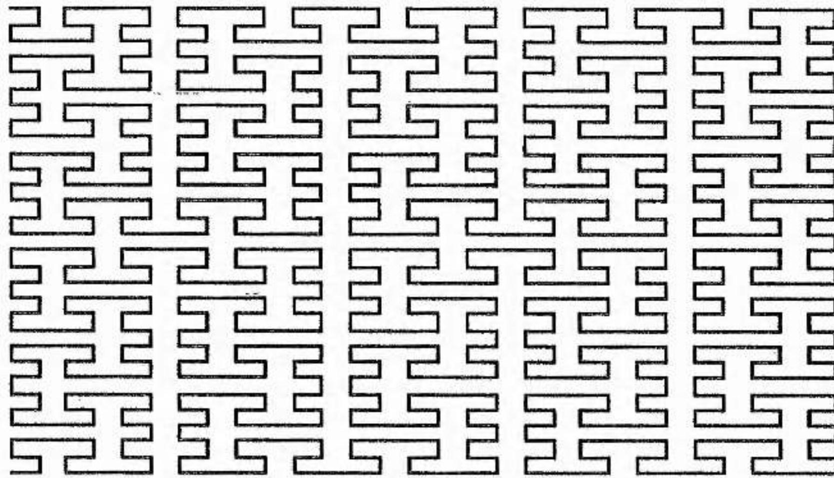
Convert the pair of murray integers (i.e., x", y") into the original fixed base number pair (x,y). The whole scheme is given in Figure 2.2.



(a) x-radix 7 and y-radix 4.



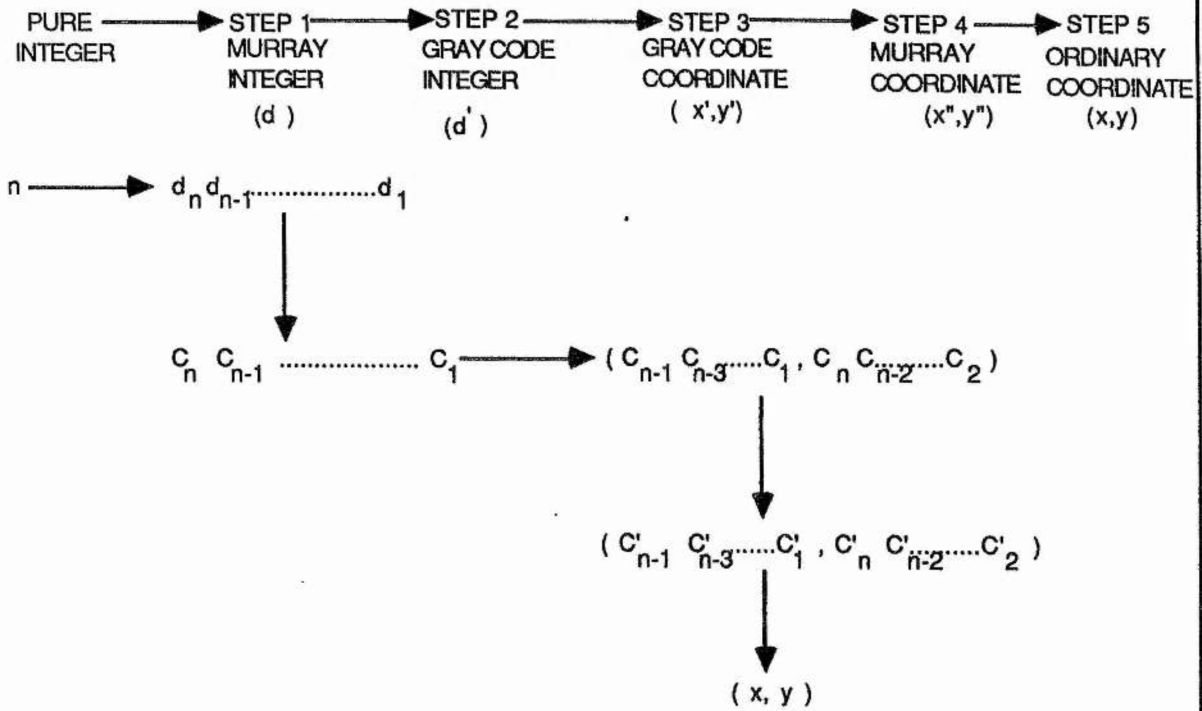
(b) x-radices 1 3 3 and y-radices 3 3 2.



(c) x-radices 2 3 5 and y-radices 3 5 2.

Figure 2.1. Murray polygons with even radices.

Figure 2.2. Murray Transformation from n to (x,y) .



Where

$d_i = n \text{ rem } r_i$ the new value of n will be equal to $(n \text{ div } r_i)$
Here i will start from 1 and will range upto n .

$$C_i = d_i \quad \text{if } \sum_{i+1}^{n} d_i \text{ is even,}$$

$$r_i - 1 - d_i \quad \text{otherwise.}$$

$$C'_i = C_i \quad \text{if } C_{i+2} + C_{i+4} + \ldots\ldots + .. \text{ is even}$$

$$r_i - 1 - C_i \quad \text{otherwise.}$$

$$x = ((\ldots\ldots(C'_{n-1} * r_{n-3} + C'_{n-3})* r_{n-5} + C'_{n-5})* r_{n-7} + \ldots\ldots + C'_1)$$

and finally,

$$y = ((\ldots\ldots(C'_n * r_{n-2} + C'_{n-2})* r_{n-4} + C'_{n-4})* r_{n-6} + \ldots\ldots + C'_2)$$

The resulting dimensions of the bounding rectangle M by N are given by

$$M = r_1 * r_3 * r_5 \,........\, * r_{n-1} \qquad \text{(product of odd radices)}$$

$$N = r_2 * r_4 * r_6 \,........\, * r_n \qquad \text{(product of even radices).}$$

Examples of several murray scans for different rectangles are given in Figure 2.3. The figure also shows how the dimensions of each sub-tile can be found by considering pairs of adjacent radices. The pair r_1, r_2 giving the x and the y dimensions of the smallest basic tile, $r_1 * r_3$, $r_2 * r_4$ the next and so on. These examples also highlight the effect of the order of the radices. A radix value of 1 can be used to force movement in a particular direction. For instance for any tile pair r_n, r_{n-1} if the least significant radix, namely the 'x radix' r_{n-1} has value 1 then all steps are forced to occur in the y direction, we will refer to this as a linear vertical murray scan, as shown in the basic tile of Figure 2.3b(a,e). Similarly movement can be restricted to the x direction by making the 'y radix' take value 1 as shown in Figure 2.3b(b,d) this is referred as a linear horizontal murray scan. Further when the dimensions of the bounding rectangle cannot be factorised into an equal number of factors then we can use a radix of value 1. The radices are packed with additional dummy radices of value 1. This is illustrated in Figure 2.3b. (c,d). It should be noted that it is advantageous that the fundamental algorithm works with a radix value of 1.

As mentioned earlier the restriction on the radices being odd can be relaxed for the first (normally x) radix and the last (normally y) radix. The effect of the first radix being even is to give the *basic* tile an even dimension. If the last radix is even then the number of horizontal scans of the largest tile is now even. For example, let us consider a rectangle whose end points are marked as 1,2,3, and 4 (see below). The path of the murray scan is also shown.

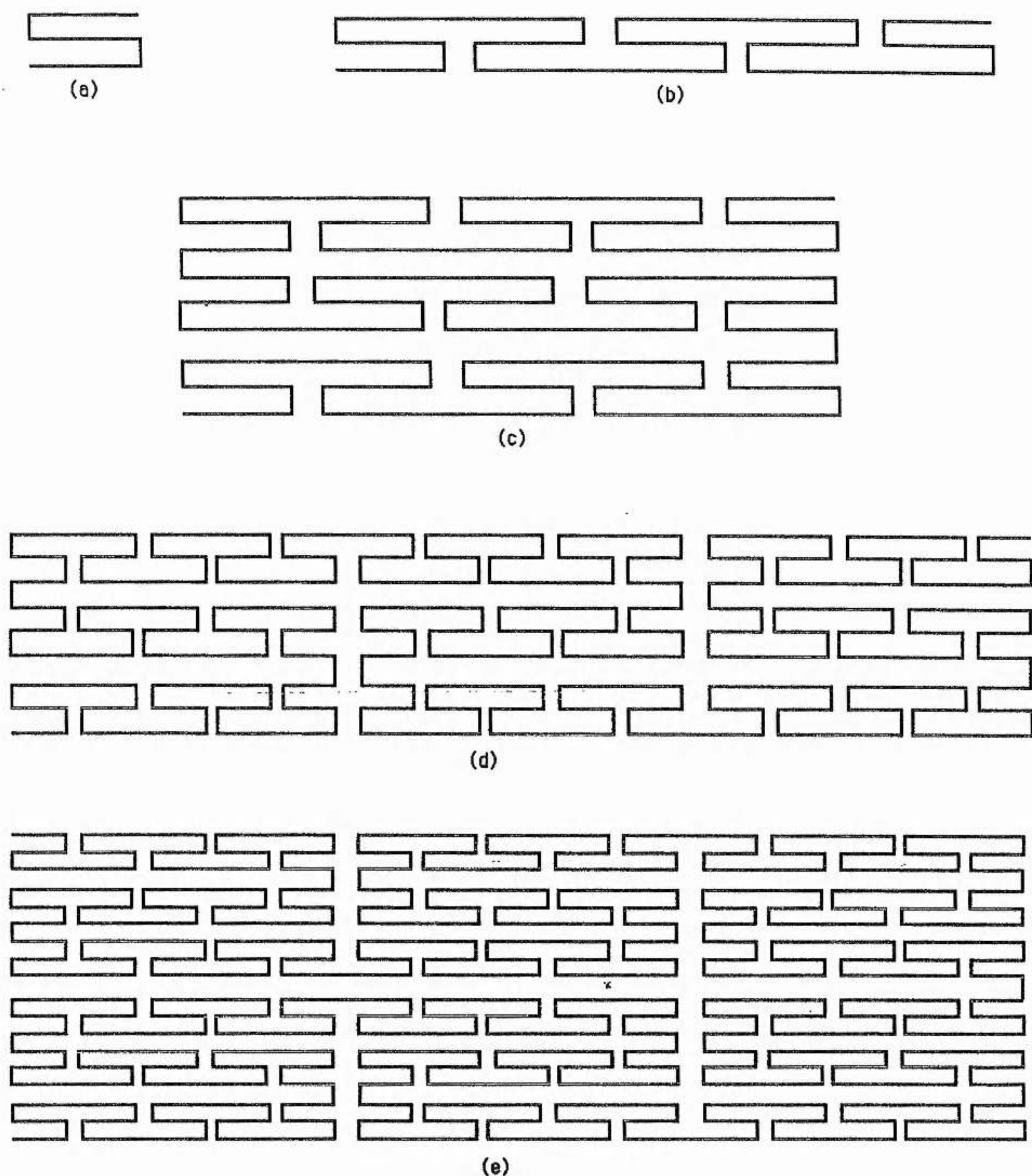


Figure 2.3a. Building steps for a murray polygons with x-radices 5 5 3 and y-radices 3 3 2.
(a) represent the smallest tile of size 5*3.
(b) represent the smallest block having 5 tiles of size 5*3.
(c) represent the next block having 3 smallest block of size 5*3*5.
(d) represent the next consecutive block having 3 blocks of size 5*3*5*3.
(e) represent the complete polygon having 2 blocks of size 5*3*5*3*3 or having 5*3*3*2 tiles each of size 5*3.

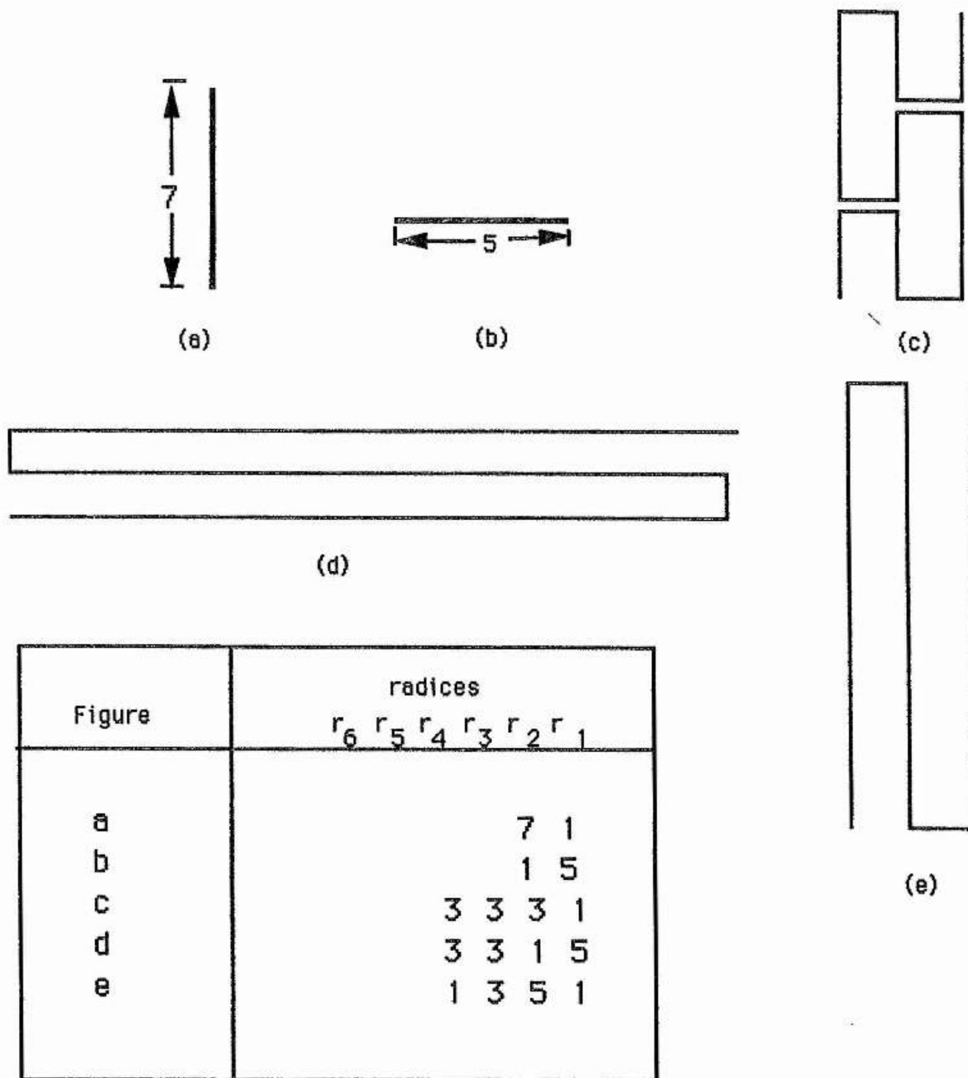


Figure	radices r_6	r_5	r_4	r_3	r_2	r_1
a					7	1
b					1	5
c			3	3	3	1
d			3	3	1	5
e			1	3	5	1

Figure 2.3b. The effect of radix value 1 on the direction of the scan.

4 3 1 2

Let r_1 and r_2 be the two radices, where r_1 corresponds to the x-part and r_2 corresponds to the y-part. In another way we can say that the x-radices represent the number of columns and the y-radices represent the number of rows. Now if we follow the linear murray path going from 1 to 3, we need an odd number of rows i.e., y-radices should be odd, whatever the x-radices, and to reach 4 we need an even number of rows i.e., y-radices should be even, whatever the x-radices. Further, if r_1, r_2,r_3,r_4 are the radices, then here r_1 and r_4 can be even but r_2 and r_3 should be odd. The radix r_2 cannot be even because the end point where we have to join the next tile is 3. In the case of r_3, consider the tile of size r_1*r_2 as a one single column . Now to reach the point 3 we need an odd number of tiles each of size r_1*r_2, hence the radices r_3 should be odd.

Note : each radices tells how many times the previous polygon has to be repeated.

2-2.4 Mixed Scans :

Cole (1988a) modified the murray polygon algorithm to allow switching between transformations in different parts of the scan. Cole used murray scans to produce bilevel hardcopy from grey scale image data giving results similar to half-toning and found that mixed scans are useful in reducing the patterns caused by the standard murray scan. The main reason for

that is the standard murray scans have only four possible orientations, whereas mixed scans have eight possible orientations. He also combines this idea with the scan patterns described by Griffiths(1986) to give mixed Griffiths and murray scans (see Cole(1988a)). An example of a simple switch between basic horizontal and vertical murray scans is given in Figure 2.4.

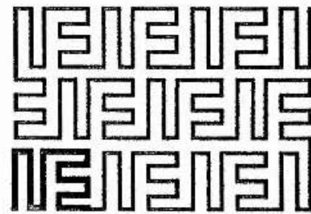


Figure. 2.4 Mixed scan (see Cole(1988a)).

2-2.5 Some Lemmas :

In the rest of this thesis only murray integers in which all of the corresponding radices are odd i.e., murray-o integers, will be considered. Further it has been assumed that d is the murray-o integer, d' is the gray code integer, the point (x', y') related to integer gray code scale axis, and the point (x", y") is the murray integer obtained by converting x' and y' separately. Other notation has been defined in Figure 2.2.

Lemma 1:

Suppose

(1). $d_{i+1}+d_{i+2}+\ldots\ldots\ldots\ldots\ldots\ldots+d_n$ is even and

(2). $C_{i+2}+C_{i+4}+\ldots\ldots\ldots\ldots\ldots\ldots+C_n$(or C_{n-1}) is also even, Then

$C'_i = d_i$ if

Case 1. i is even and

$$d_{i+1}+d_{i+3}+\dots\dots\dots+d_{n-1} \text{ is even.}$$

Case 2. i is odd and

$$d_{i+1}+d_{i+3}+\dots\dots\dots+d_{n} \text{ is even.}$$

Where d_i, C_i , and C'_i are defined in Figure 2.2.

Proof :

For the given conditions, we have

$$C_i = d_i \text{ and } C'_i = C_i$$

which implies $C'_i = d_i$ for i to be odd or even.

Case 1. i is even. From (2) we have

$$C_{i+2}+C_{i+4}+\dots\dots\dots+C_n = \text{even}$$

given $C_i = d_i$, this implies $d_{i+2}+d_{i+4}+\dots\dots\dots+d_n = \text{even}$ (3)

Equation (1) can be written as

$$(d_{i+2}+d_{i+4}+\dots\dots\dots+d_n) + (d_{i+1}+d_{i+3}+\dots\dots\dots+d_{n-1}) = \text{even}$$

using (3) we get,

$$\text{even} + (d_{i+1}+d_{i+3}+\dots\dots\dots+d_{n-1}) = \text{even}$$

or

$$(d_{i+1}+d_{i+3}+\dots\dots\dots+d_{n-1}) = \text{even} .$$

Case 2. i is odd. From (2) we have

$$C_{i+2}+C_{i+4}+\ldots\ldots\ldots\ldots\ldots+C_{n-1} = \text{even}$$

given $C_i = d_i$, this implies $d_{i+2}+d_{i+4}+\ldots\ldots\ldots+d_{n-1} = \text{even}$ (3)

Equation (1) can be written as

$$(d_{i+2}+d_{i+4}+\ldots\ldots\ldots+d_{n-1}) + (d_{i+1}+d_{i+3}+\ldots\ldots\ldots+d_n) = \text{even}$$

using (3) we get,

$$\text{even} + (d_{i+1}+d_{i+3}+\ldots\ldots\ldots+d_n) = \text{even}$$

or

$$(d_{i+1}+d_{i+3}+\ldots\ldots\ldots+d_n) = \text{even}$$

Hence proved.

Lemma 2 :

Suppose

(1). $d_{i+1}+d_{i+2}+\ldots\ldots\ldots\ldots\ldots+d_n$ is odd and

(2). $C_{i+2}+C_{i+4}+\ldots\ldots\ldots\ldots\ldots+C_n$(or C_{n-1}) is also odd. Then

$$C'_i = d_i \text{ if}$$

Case 1. i is even and

$$(r_{i+2}+r_{i+4}+\ldots\ldots+r_n) + (d_{i+1}+d_{i+3}+\ldots\ldots+d_{n-1}) - (n/2 - i \text{ div } 2) \text{ is even.}$$

Case 2. i is odd and

$$(r_{i+2}+r_{i+4}+\ldots\ldots+r_{n-1}) + (d_{i+1}+d_{i+3}+\ldots\ldots+d_n) - (n/2 - (i+1) \text{ div } 2) \text{ is even.}$$

Where d_i, C_i , and C'_i are defined in Figure 2.2.

Proof :

From the given conditions we have,

$$C_i = r_i - d_i - 1 \quad \text{and} \quad C'_i = r_i - C_i - 1.$$

Substituting C_i in latter case we get,

$$C'_i = r_i - (r_i - d_i - 1) - 1$$

or $$C'_i = d_i$$

Case 1. if i is even. Substituting $C_i = r_i - d_i - 1$ in equation (2) we get,

$$(r_{i+2} - d_{i+2} - 1) + (r_{i+4} - d_{i+4} - 1) + \dots\dots\dots + (r_n - d_n - 1) = \text{odd}$$

$$(r_{i+2}+r_{i+4}+\dots\dots+r_n) - (d_{i+2}+d_{i+4}+\dots\dots+d_n) - (n/2 - i \text{ div } 2) = \text{odd} \quad (3)$$

Equation (1) can be written as

$$(d_{i+2}+d_{i+4}+\dots\dots\dots+d_n) + (d_{i+1}+d_{i+3}+\dots\dots\dots+d_{n-1}) = \text{odd} \quad (4)$$

Using equation (4), equation (3) can be written as,

$$(r_{i+2}+r_{i+4}+\dots\dots+r_n) - \text{odd} + (d_{i+1}+d_{i+3}+\dots\dots+d_{n-1}) - (n/2 - i \text{ div } 2) = \text{odd}$$

or $$(r_{i+2}+r_{i+4}+\dots\dots+r_n) + (d_{i+1}+d_{i+3}+\dots\dots\dots+d_{n-1}) - (n/2 - i \text{ div } 2) = \text{even}$$

Case 2. if i is odd. Here equation (3) can be written as,

$$(r_{i+2}+r_{i+4}+\dots\dots+r_{n-1}) - (d_{i+2}+d_{i+4}+\dots\dots+d_{n-1}) - (n/2 - (i+1) \text{ div } 2) = \text{odd}$$

Using equation (1) in the above equation we get the required result i.e.,

$$(r_{i+2}+r_{i+4}+\dots\dots+r_{n-1}) + (d_{i+1}+d_{i+3}+\dots\dots+d_n) - (n/2 - (i+1) \text{ div } 2) = \text{even}.$$

Hence proved.

Lemma 3 :

Suppose

(1). $d_{i+1}+d_{i+2}+\dots\dots+d_n$ is even and

(2). $C_{i+2}+C_{i+4}+\dots\dots+C_n$(or C_{n-1}) is odd then

$C'_i = r_i - d_i - 1$ if

Case 1. i is even,

$d_{i+1}+d_{i+3}+\dots\dots+d_{n-1} = \text{odd}.$

Case 2. i is odd,

$d_{i+1}+d_{i+3}+\dots\dots+d_n = \text{odd}.$

Where d_i, C_i , and C'_i is defined in Figure 2.2.

Proof :

Proof is very similar to Lemma 1.

Lemma 4 :

let

(1). $d_{i+1}+d_{i+2}+\dots\dots+d_n$ is odd and

(2). $C_{i+2}+C_{i+4}+\dots\dots+C_n$(or C_{n-1}) is even then

$C'_i = r_i - d_i - 1$ if

Case 1. i is even and

$$(r_{i+2}+r_{i+4}+\dots\dots\dots+r_n) + (d_{i+1}+d_{i+3}+\dots\dots\dots+d_{n-1}) - (n/2 - i \text{ div } 2) \text{ is odd.}$$

Case 2. i is odd and

$$(r_{i+2}+r_{i+4}+\dots\dots+r_{n-1}) + (d_{i+1}+d_{i+3}+\dots\dots+d_n) - (n/2 - (i+1) \text{ div } 2) \text{ is odd.}$$

Where d_i, C_i , and C'_i are defined in Figure 2.2.

Proof :

Proof is very similar to Lemma 2.

Theorem 1:

If d is the murray-o integer and the point (x", y") are the corresponding murray coordinates as defined in Figure 2.2, then,

$$C'_i = d_i \quad \text{if} \quad d_{i+1} + d_{i+3} + \dots\dots\dots\dots + (d_{n-1} \text{ or } d_n) \quad = \text{even,}$$

$$r_i - d_i - 1 \qquad \text{otherwise.}$$

Proof :

From lemma 1 and 2 we have,

$C'_i = d_i$ if

(I) i is even,

$$d_{i+1}+d_{i+3}+\dots\dots\dots\dots+d_{n-1} = \text{even, and}$$

$$(r_{i+2}+r_{i+4}+\dots+r_n) +(d_{i+1}+d_{i+3}+\dots+d_{n-1}) -(n/2 - i \text{ div } 2) = \text{even}$$

(II) i is odd,

$d_{i+1}+d_{i+3}+\ldots\ldots\ldots\ldots+d_n$ = even,

$(r_{i+2}+r_{i+4}+\ldots+r_{n-1}) +(d_{i+1}+d_{i+3}+\ldots+d_n) -(n/2 - i \text{ div } 2 +1)$ =even.

Here the value for d_i can be odd or even, and the value for r_i will be odd for each i, this implies the value for $(r_i -d_i -1)$ will be even if d_i is even, otherwise odd, since $(r_i -1)$ is even. The above statement can be restated as,

$C'_i = d_i$ if $d_{i+1}+d_{i+3}+\ldots\ldots\ldots\ldots +(d_{n-1}$ or $d_n)$ = even ------------ (A)

Similarly from lemma 3 and 4 we have,

$C'_i = r_i - d_i -1$ if

(I) i is even,

$d_{i+1}+d_{i+3}+\ldots\ldots\ldots\ldots+d_{n-1}$ = odd

$(r_{i+2}+r_{i+4}+\ldots\ldots+r_n) + (d_{i+1}+d_{i+3}+\ldots\ldots+d_{n-1}) - (n/2 - i \text{ div } 2)$ = odd.

(II) i is odd,

$d_{i+1}+d_{i+3}+\ldots\ldots\ldots\ldots+d_n$ = odd

$(r_{i+2}+r_{i+4}+\ldots+r_{n-1}) +(d_{i+1}+d_{i+3}+\ldots+d_n) -(n/2 - i \text{ div } 2 +1)$ = odd.

Here again we can say that the term $(r_i -d_i -1)$ will be odd if d_i is odd and using this we get,

$C'_i = r_i - d_i -1$ if $d_{i+1}+d_{i+3}+\ldots\ldots\ldots+(d_{n-1}$ or $d_n)$ = odd ---------- (B)

Hence proved from result (A) and (B).

2-2.6 Implementation of murray scan :

The original implementation by Cole is given in I) and his fast murray scan is given in II). Both the programs are coded in PS-algol and C.

I) Original implementation :

Only main procedures are given. All the steps are given in section 2-2.3.1. The murray integer is held in an array of integers and this value is incremented to move from vertex to vertex on the scan. The steps for drawing murray curves from a point (say nth) to (x,y) are,

(*Note : The nth point is not the same as we use in murray digits)*

1. Convert n to the equivalent murray integer with the given murray radices
2. Convert this integer to the equivalent Gray coded integer
3. Split this integer into x and y,
4. De-Gray code x and y parts using alternate digits for each part,
5. Convert x and y back to ordinary integers giving (x,y),
6. Increment murray integer by one,
7. Repeat steps 2 to 6 while n<= N*M-1, where N and M are the dimensions for a given rectangle.

Conversely, to convert from (x,y) to n, the above steps are reversed, starting from step 5. The conversion from a murray integer to n is given in *convert.from.murray* procedure.

```
! Input parameters are an integer and an array of radices
! Output is the corresponding array of murray digits
let convert.to.murray = proc( int n; *int radices -> *int )
begin
        let murray.int = vector 1 :: upb( radices ) of 0
        let i := 1
        while n ~= 0 do
        begin
                murray.int( i ) := n rem radices( i )
                n := n div radices( i )
                i := i + 1
        end
        murray.int
end

! Input parameters are murray integer and radices arrays
! Output is the corresponding integer
let convert.from.murray = proc( *int murray.int,radices -> *int )
begin
        let top = upb( murray.int )
        let n := murray.int( top )
        for i = top - 1 to 1 by -1 do
            n := n * radices( i ) + murray.int( i )
        n
end

! Input parameters are murray integer and radices arrays
! The murray integer array is incremented by 1
let next.murray = proc( *int murray.int, radices )
begin
        let i := 1
        while murray.int( i ) = radices( i ) -1 do
        begin
                murray.int( i ) := 0
                i := i + 1
        end
        murray.int( i ) := murray.int( i ) + 1
end
```

The following procedure gray.code may be used both for conversion to and from grey coded murray integers.

```
! Input parameters are murray.int and radices arrays
! murray.int is converted to the Gray code equivalent
let gray.code = proc( *int murray.int, radices )
begin
        let top = upb( murray.int )
        let parity :=(  ( murray.int( top ) rem 2 ) = 1 )
        for i = top - 1 to 1 by -1 do
        begin
                if parity do
                        murray.int( i ) := radices( i ) -1 - murray.int( i )
                if ( murray.int( i ) rem 2 = 1 ) do
                        parity := ~parity
        end
end

! This procedure takes the array of digits splits it into x and y parts.
! Input is a murray integer
! Output is a structure holding the x and y murray integer arrays
structure coords( *int a,b )
let split.x.y = proc( *int murray.int -> pntr )
begin

        let top = upb( murray.int )
        let x = vector 1 :: top div 2 of 0
        let y = vector 1 :: top div 2 of 0
        let i := 1
        for j = 1 to top - 1 by 2 do
        begin
                x( i ) := murray.int( j )
                y( i ) := murray.int( j + 1 )
                i := i + 1
        end
        coords( x,y )
end
```

Some Improvements of The Original Implementation:

Using ***Theorem 1***, the number of steps given in the original implementation can be reduced, which can increase the efficiency for the scan.The steps of transformation will now be,

1. Convert n to the equivalent murray integer with the given murray radices

   (i.e., *convert.to.murray)*

2. Convert this integer to the equivalent murray coordinates (x,y),
3. Convert x and y back to ordinary integers giving (x,y),
4. Increment murray integer by one,
5. Repeat steps 2 to 4 while n<= N*M-1, where N and M are the dimensions for a given rectangle.

II) A Faster murray scan algorithm :

The original implementations required an improvement in efficiency for large complete scans. The parts of the first implementation that slow the algorithm down are the conversion to and from murray integers to pure integers and the transformation using the gray-code procedure. In the improved case also the conversion from murray integer to murray coordinates and (x,y) to ordinary integers is time consuming. If we examine the murray curve , we see that, a point inside a rectangle has only four possible ways to move. Either it can go left or right or up or down. Each time only one of the coordinate is going to be incremented or decremented by one unit. Now the only problem is to find which coordinate is going to be incremented or decremented.

Suppose

$$d = d_n d_{n-1} \ldots\ldots\ldots\ldots d_3 d_2 d_1.$$

be a murray digits with radices, r_i ($i = 1\ldots\ldots n$) such that $0 <= d_i <= r_i - 1$ where m is any integer and n is equal to 2m. Let p_i be the parity of the sum of the digits $d_{i+1}, d_{i+2}, \ldots\ldots, d_n$. That is, p_i has value **true** if this parity is even and **false** otherwise. Let $C_n, C_{n-1}, \ldots\ldots, C_1$ be the equivalent Gray code integer, where C_i is equal to d_i if p_i is **true** otherwise $r_i - 1 - d_i$ if p_i is **false.** Also if r_i is odd the parity of the new digit d_i is unchanged in both cases (the parity of $r_i - 1 - d_i$ where $r_i - 1$ will always be even and even - d_i can be even or odd, dependand upon the digit d_i itself), so the values of p_j for $j < i$ are unchanged. The back Gray code transformation on d_i depends on whether i is itself odd or even . If i is odd then the digit d_i belongs to the x coordinate otherwise it is part of the y coordinate. The back Gray code of d_i depends on the parity of the sum of digits $d_{i+2}, d_{i+4}, d_{i+6}, \ldots\ldots\ldots, d_k$ (where k is equal to n, if i is even otherwise k is n-1) and is d_i if this parity is **true** and $r_i - 1 - d_i$ if it is **false**.

Further the same result can be obtained by using***theorem 1.*** Here the transformed digit d_i is equal to d_i if the parity of sum of digits $d_{i+1}, d_{i+3} \ldots\ldots\ldots$ is **true** otherwise, $r_i - d_i - 1$ if it is **false**. The rule for the total change in a given digit can now be summarised as follows. Define q_i to have the value **true** if the parity of sum of digits $d_{i+1}, d_{i+3} \ldots\ldots\ldots$ is even and **false** otherwise. The digit d_i remains unaltered if q_i is **true** and is replaced by $r_i - 1 - d_i$ if q_i is **false.**

Consider now the case of a murray integer about to be increased by one. This will cause a change in parity of digit d_i. Either d_i is the first digit or carry has taken place in one or more positions and all the digits to the right of

d_i are changed from r_j -1 to 0. Now 0 and r_i-1 are both even hence the only digit to change parity is d_i. Thus the parities q_{i-1}, q_{i-3}, ... will have changed. We now see that when a murray integer increases by one there is only one digit which changes parity and its position determines whether a change has occurred in x or y, and also which q values to change.

The only remaining information to determine is whether the change is +1 or -1. The subscript 'i' identifies the part under consideration, since all odd digits corresponds to the x part and all even digits corresponds to the y part. It should be noted that the x digits are selected from among the d-digits and that from the general theory two successive x digits selected in this way differ by only 1. The only problem now is to determine out of these two digits which one in numerically greater, that is whether the x increment is to be positive or negative. It follows from the general theory that this can be determined by the parity digit p_{i+1}. If p_{i+1} is *true* then the x increment is 1 otherwise it is -1. Similarly for a step in the y direction.

The faster algorithm thus only requires the murray digits arrray to be incremented by 1 and a parity array to be maintained. The steps for the faster algorithm are thus ,

1. Increment the array digit by 1,
2. if the ith digit changes then change the parities of p_i , p_{i-2}, p_{i-4}......,
3. choose the x or y direction to be incremented according as i is odd or even,
4. increment the chosen direction by 1 or -1 according as p_{i+1} has value **true** or **false.**

Procedures to implement this algorithm are as follows.

The procedure to give the next murray integer is same as next.murray procedure given above except that it returns the index of the leftmost digit to change in the murray integer.

```
! Input parameters are murray integer (i.e. d) and radices arrays (i.e., r)
! The murray integer array is incremented by 1
! The index of the leftmost digit to change is returned
let increment := proc(*int d,r; int i -> int)
nullproc
increment := proc(*int d,r; int i -> int)
if d(i) < r(i) - 1 then { d(i) := d(i) + 1 ; i}
                    else { d(i) := 0; increment(d,r,i+1) }
```

The parity of the sums of alternate digits in a murray integer d to the left of the index i is held in an array p of truth values. The following procedure changes the truth value of p(i) and every other entry to the right of it.

```
! Input parameters are the boolean array of parities
! and the index of the leftmost digit to change
let change.parity = proc(*bool p ; int start)
for i = start to 1 by -2 do p(i) := ~p(i)
```

Finally the change in x or y is calculated by using the procedure step as defined follows,

```
! Input parameters are the parity array q
! and a digit position i
let step = proc(*bool q; int i -> int)
if q(i+1) then 1 else -1

if i rem 2 = 1 then x := x+step(q,i) else y := y+step(q,i)
```

Note that the counting is now in pure murray integers rather than gray coded murray integers.

2-2.7 Hardware Implementation :

Following on from the ideas used in the implementation of the fast algorithm outlined in the previous section Cole(1988b) suggests some hints on hardware implementation. The basic suggestion is that the function of the array of integers holding the murray integer is taken over by a bank of shift registers. Each register would have a capacity corresponding to the radices selected and be initially set with value 1. If a register is cleared by a shift operation it then resets to zero and forces a shift in the next register. If the register does not clear then the parity of the register number will identify whether the movement is in the x or y direction. There would also be a number of parity bits which can be toggled appropriately and will determine if the step is -1 or +1.

2-2.8 Three-Dimensional Cartesian Coordinate :

The three-dimensional Cartesian (rectangular) coordinate systems consists of a reference point, called the ***origin*** and three mutually perpendicular lines passing through the origin, called the ***axis.*** These mutually perpendicular lines are labelled the x, y, and z coordinate axis (see Figure 2.5). To every point P there corresponds uniquely a set of three numbers [x,y,z], and conversely to every set of three numbers, positive or negative, there corresponds a unique point. For the n-dimensional case there will be n mutually perpendicular axis.

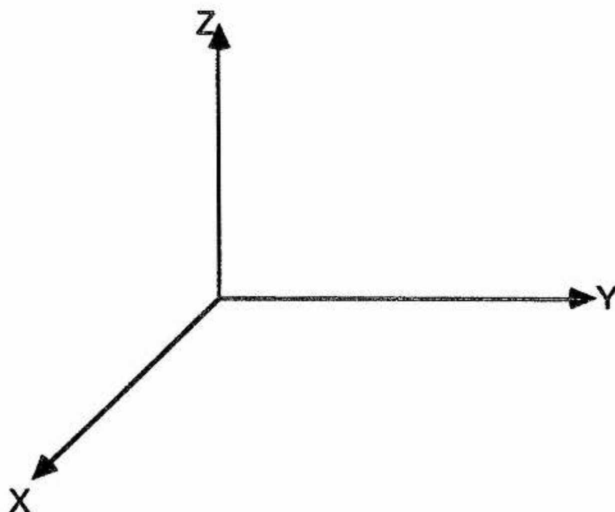


Figure 2.5.

2-2.8.1 Extension To Three-Dimensional And n-Dimensional Murray Polygons :

Here we are going to extend the idea of one dimensional and two dimensional murray polygons, which are described in previous section 2-2.6, to three dimensional and n dimensional murray polygons. The algorithms which have been described in section 2-2.6 are used. The only change which will come to the previous algorithms is the addition of more digits to the murray digits. These additional digits will corresponds to the other axes. In the case of three dimension the additional digits will corresponds to the z-axis. The methods depends upon the original implementation by Cole and his fast version for one-dimensional and two-dimensional murray polygons. The methods are discussed below.

2-2.8.1.1 Method One :

Let $d = d_n d_{n-1} \ldots . d_2 d_1$ (where n is a multiple of 3) be a murray integer with radices, $r_i(i=1,2,.......,n)$ such that, for each i we have $0 \leq d_i \leq r_i-1$.

Where,

digits $d_1d_4d_7..............d_{n-2}$ belong to the x coordinates of the box,
digits $d_2d_5d_8..............d_{n-1}$ belong to the y coordinates of the box and
digits $d_3d_6d_9..............d_n$ belong to the z coordinates of the box.

The dimensions for the bounding box are assumed to be 'l', 'b', and 'h' and are given as,

$$l = r_1 * r_4 * r_7 * .... * r_{n-2}$$

$$h = r_2 * r_5 * r_8 * .... * r_{n-1}$$

$$b = r_3 * r_6 * r_9 * .... * r_n$$

The triple r_1, r_2, r_3 are the x, y and the z dimensions of the smallest box, r_1*r_4, r_2*r_5, r_3*r_6 are the x, y and the z dimensions for the next box which has $r_4*r_5*r_6$ boxes of size $r_1*r_2*r_3$, and so on. The dimension 'b' represents the number of planes each of size l*h, and parallel to the XY-plane. The planes will range from '0' to 'b-1' i.e. value for the z coordinate. The stages for drawing a 3D murray curve are outlined as follows,

Convert this murray integer to a Gray code integer

$$d' = c_n c_{n-1} \cdots c_2 c_1.$$

where, $c_i = d_i$ if the sum of all its more significant digits is even,

$= r_i - 1 - d_i$ otherwise.

Split the Gray code number into parts x', y', and z' as below

$$x' = c_{n-2} .... c_4 c_1$$

$$y' = c_{n-1} .... c_5 c_2$$

$z' = c_n \ldots\ldots\ldots\ldots c_6 c_3$

Convert Gray code x', y', and z' separately back into murray integers

(x", y", z") given as

$(x'',y'',z'') = (c'_{n-2} \ldots\ldots\ldots c'_1 , c'_{n-1} \ldots\ldots\ldots c'_2 , c'_n \ldots\ldots\ldots c'_3)$ where

$c'_i = c_i$ if the sum of all its more significant digits is even,

$= r_i - 1 - c_i$ otherwise.

Convert the pair of murray integers (i.e., x", y", z") into the original fixed base number pair (x,y,z). The steps of transformations from n to (x,y,z) can now be summarised as,

1. Convert n to the equivalent murray integer with the given murray radices
2. Convert this integer to the equivalent Gray coded integer
3. Split this integer into x, y, and z,
4. De-Gray code x, y, and z parts using alternate digits for each part,
5. Convert x", y", and z" back to ordinary integers giving (x,y,z),
6. Increment murray integer by one,
7. Repeat steps 2 to 6 while n<= l*b*h-1, where 'l', 'b', and 'h' are defined above.

Conversely, to convert from (x,y,z) to n, the above steps are reversed, starting from step 5.

Now for the faster algorithm we can define few lemmas.

Lemma 5 :

Suppose

(1). $d_{i+1}+d_{i+2}+$...........................$+d_n$ is even and

(2). $c_{i+3}+c_{i+6}+c_{i+9}+$...........................$+c_n$(or c_{n-2} or c_{n-1}) is even then

$c'_i = d_i$ if

$d_{i+1}+d_{i+2}+ d_{i+4}+d_{i+5}+$........................ = even.

Where d_i, c_i , and c'_i are defined above.

Proof:

Proof is very similar to the lemmas discussed for the two-dimensional case.

Lemma 6 :

Suppose

(1). $d_{i+1}+d_{i+2}+$...........................$+d_n$ is even and

(2). $c_{i+3}+c_{i+6}+c_{i+9}+$...........................$+c_n$(or c_{n-2} or c_{n-1}) is odd, then

$c_i = r_i - d_i -1$ if

$d_{i+1}+d_{i+2}+ d_{i+4}+d_{i+5}+$........................ = odd.

Where d_i, c_i , and c'_i are defined above.

Proof:

Proof is very similar to the lemmas discussed for the two-dimensional case.

Similarly we can define two more lemmas where in the first case the condition (1) is odd and (2) is even and in the second case the condition (1) is odd and (2) is also odd. The proof is similar to the one discussed for two-dimensional case.

Theorem 2:

If d is a murray integer and the point (x",y",z") are the corresponding murray coordinates as defined above then,

$c'_i = d_i$ if $d_{i+1} + d_{i+2} + d_{i+4} + d_{i+5} + \dots\dots\dots\dots$ is even

$r_i - 1 - d_i$ otherwise.

Proof:

Using these lemmas discussed above the theorem 2 can be proved. The proof is similarly to ***Theorem 1.***

Using ***Theorem 2*** the number of steps can be reduced. The steps of transformation will now be,

1. Convert n to the equivalent murray integer with the given murray radices
2. Convert this integer to the equivalent murray coordinates (x,y,z),
3. Convert x, y, and z back to ordinary integers giving (x,y,z),

4. Increment murray integer by one,

5. Repeat steps 2 to 6 while n<= l*b*h-1, where 'l', 'b', and 'h' are defined above.

2-2.8.1.2 Second Method :

This method is the extension of the fast murray scan given by Cole for one-dimensional and two dimensional space. Here a point inside a box has six possible ways to move. It can go either (up or down) or (left or right) or (front or back). Each time only one of the coordinate is going to be incremented or decremented. Which coordinate is going to be incremented or decremented can be determined by the parity changes. This is briefly discussed below,

The idea is similar to the one discussed in *section 2-2.6(II)*

Suppose

$$d = d_n d_{n-1} \ldots\ldots\ldots\ldots d_3 d_2 d_1.$$

is a murray integer with radices, r_i ($i = 1\ldots\ldots n$) such that $0 <= d_i <= r_i - 1$ where m is any integer and n is equal to 3m. Let p_i be the parity of the sum of the digits $d_{i+1}, d_{i+2}, \ldots\ldots\ldots, d_n$. That is, p_i has value **true** if this parity is even and **false** otherwise . The digit d_i remains unaltered if p_i is **true** and is replaced by $r_i - 1 - d_i$ if p_i is **false.**

Similarly using ***theorem 2,*** the transformed digit d_i can be equal to d_i if the parity of the sum of digits $d_{i+1}, d_{i+2}, d_{i+4}, d_{i+5} \ldots\ldots\ldots$ is **true** otherwise, $r_i - d_i - 1$ if it is **false.** We can now define q_i to take the value **true** if the parity of sum of digits $d_{i+1}, d_{i+2}, d_{i+4}, d_{i+5} \ldots\ldots\ldots$ is even and **false** otherwise. The digit d_i remains unaltered if q_i is **true** and is replaced by $r_i - 1 - d_i$ if q_i is **false.**

Consider now the case of a murray integer about to be increased by one. If the leftmost digit to change is in ith position then the parities for q_i, q_{i-1}, q_{i-3} , q_{i-4}, q_{i-6}............. will change. The problem of determining the direction to be incremented or decremented can be obtained from the value of subscript i. If i belongs to 1,4,7,... then the x part, if it belongs to 2,5,8,..... then the y part, and for others we will consider the z part to change. Now the steps of the transformation will be the same as given for 1D and 2D cases, only the parity changes will alter as discussed above.

The 3D murray polygons can be build in two different ways

1) plane by plane, 2) tile by tile, as in the case of an octree.

Both ways depends upon the value for the z radices. Plane by plane scanning can be useful for scanning medical images which are in the form of consecutive planes.

Let d_6,d_5,d_4,d_3,d_2,d_1 be the murray digits with radices, r_i (i = 1....n) such that for each i, $0 <= d_i <= r_i$.

If digit r_3 has value 1 then the algorithm will scan the first plane (i.e., z=0) and then will go to second plane and so on. A change in the digit d_n implies the change in a plane number. The steps of the transformation are given in Table 2.2.

If r_3 takes any other value than 1 provided it is odd then the algorithm will scan the smallest tile (of size $r_1 * r_2$) of the first rectangle then will scan the same tile in the second rectangle and so on untill the z digit is less than r_3-1. After that it will scan the other tiles in the same way (see Figure 2.6). The steps of the transformation are given in Table 2.3. As usual the first and the last radices can be even i.e. r_1 and r_6. The radix r_6 simply means that the image has an even numbers of planes.

Murray Integers $d_6 d_5 d_4 d_3 d_2 d_1$	Integer Coordinates (x, y, z)	
000000	(0,0,0)	first plane
000001	(1,0,0)	
000002	(2,0,0)	
000010	(2,1,0)	
-------	------	
000022	(2,2,0)	
100000	(2,2,1)	second plane
100001	(1,2,1)	
------	------	
100022	(0,0,1)	
200000	(0,0,2)	Third plane
200001	(1,0,2)	
-------	-------	
-------	-------	
200022	(2,2,2)	

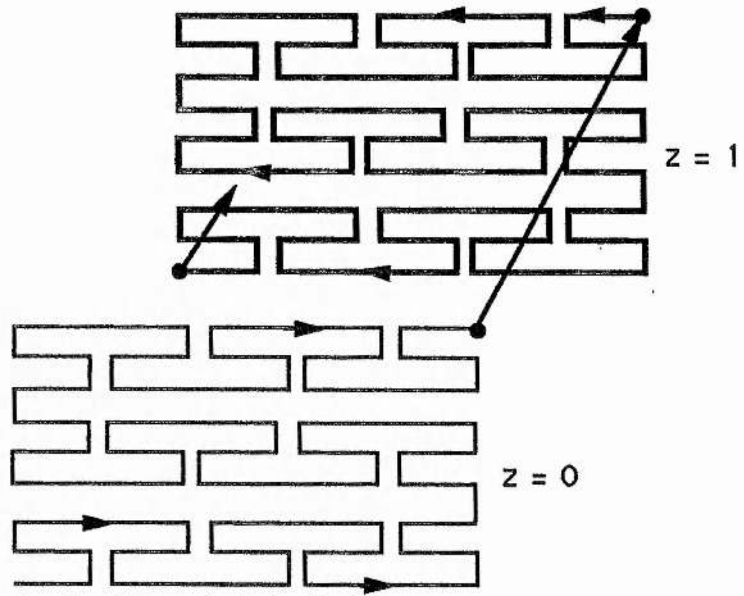
Table 2.2 :

Murray transformation from n to (x,y,z) with murray radices 3,1,1,1,3,3. The plane is same untill digit d_6 changes. Here digit d_6 takes values 0,1, and 3, which is nothing but the plane numbers i.e., z values. Other steps i.e., gray code integers, murray integers etcetra are not given.

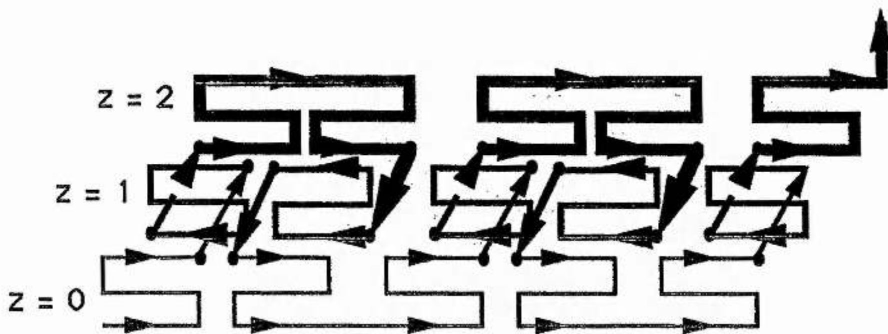
Murray Integers d_6 d_5 d_4 d_3 d_2 d_1	Integer coordinates (x,y,z)	
000000	(0,0,0)	1st plane
000001	(1,0,0)	
000002	(2,0,0)	
-------	------	
000022	(2,2,0)	
000100	(2,2,1)	2nd plane
000101	(1,2,1)	
-------	-------	
000200	(0,0,2)	3rd plane
000201	(1,0,2)	
------	------	
000222	(2,2,2)	
001000	(3,2,2)	3rd plane
001001	(4,2,2)	
-------	------	
001022	(5,0,2)	
001100	(5,0,1)	2nd plane
001101	(4,0,1)	
001102	(3,0,1)	
-------	-------	
-------	-------	1st
-------	-------	---

Table 2.3 :

Murray transformation from n to (x,y,z) with murray radices 1,1,3,3,3,3. The plane is same untill digit d_3 changes. Here digit d_3 takes values 0,1, and 3, which is nothing but the plane numbers i.e., z values. Other steps i.e., gray code integers, murray integers etcetra are not given. Further, if we want to increase the planes we can consider r_6 greater than one.



(a)



(b)

Figure 2.6. Scanning patterns due to 3-dimensional murray scan
(a) an image is scanned in plane by plane order,
(b) an image is scanned in tile by tile fashion.
(Dark colour has used to show the different planes).

The same algorithms can also be used for n-dimensional space. The only change will be in the addition of the digits for additional axes and the parity changes. The steps for an n-dimensional faster algorithm are,

1. Increment the array digits by 1,
2. if the ith digit changes then change the parities of $p_i, p_{i-1}, \ldots, p_{i-(n-2)}, p_{i-n}, p_{i-n-1}, \ldots, p_{i-(2n-2)}, \ldots$,
3. choose the $x_1, x_2, \ldots, x_{n-1}$, or the x_n direction to be incremented according as the value of i ,
4. increment the chosen direction by 1 or -1 according as p_{i+1} has value **true** or **false.**

2-2.9 Polar Murray Scan :

2-2.9.1 Polar Coordinates[Fine(1909)] :

Let *O* be a given point, and *Ox* a given directed line from *O*, where *O* is called the *pole or origin* and *Ox* is called as the *polar axis*. The *polar coordinates* of any point *P*, referred to *O* and *Ox* are given as (r, Θ), where *r* is called as *radius vector* and is equal to the length of *OP* and Θ is called the *vectorial angle* of P and is equal to the measure of the angle *xOP* (see Figure 2.7).

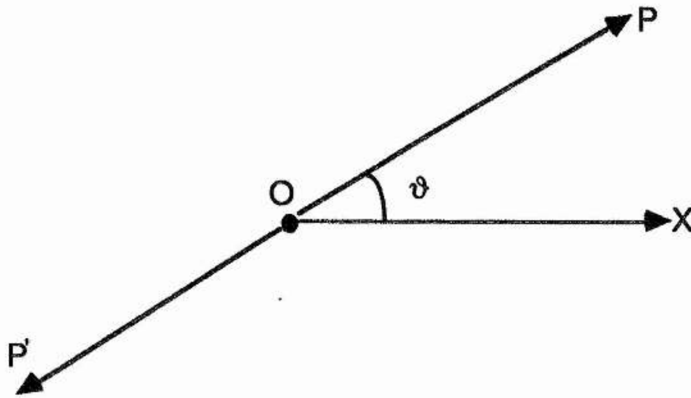


Figure 2.7.

The polar coordinates of a point are not unique. This is because the addition or subtraction of any multiple of 2Π to Θ describe the same ray as that described by Θ.

2-2.9.2 Changing Coordinate Systems :

If the polar axis *Ox* be taken as the x axis of a rectangular systems, and *Oy* as the corresponding y axis, the relations connecting the coordinates of any point *P* in the two systems are given as

$x = r \cos \Theta,$

$y = r \sin \Theta.$

$r^2 = x^2 + y^2,$

$\tan \Theta = y/x,$

$\sin \Theta = y/r,$

$\cos \Theta = x/r.$

2-2.9.3 Graphs In Polar Coordinates :

The *graphs* of *points* given in polar coordinates are obtained by taking the length *r* on the terminal line of the angle *Θ*. These lengths being measured on the line produced through the origin according as r is positive or negative. The *graph* of an equation in *r* and Θ is the collection of the points(r, Θ) of all the solutions of the equation (see Figure 2.8).

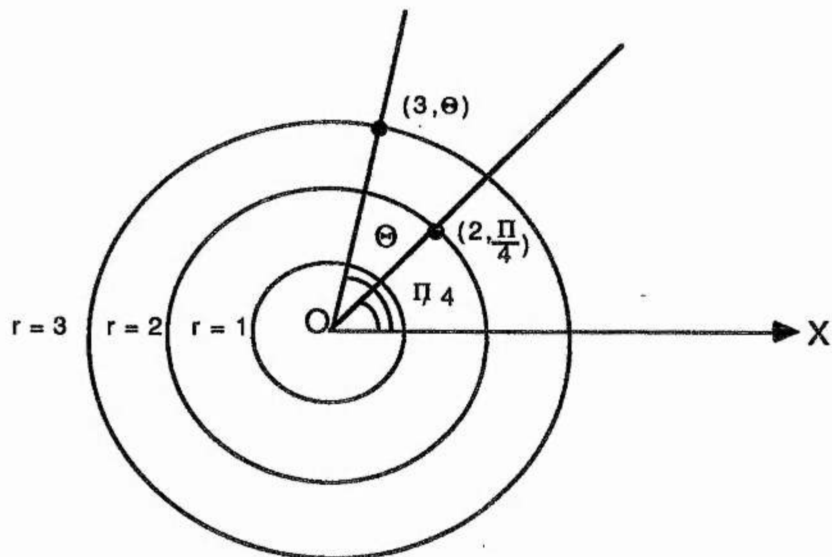


Figure 2.8

2-2.9.4 3D And Higher Dimensions Polar Coordinates:

2-2.9.4.1 Cylindrical Coordinates, Spherical Coordinates :

The only method which is not provided by Cartesian coordinates is associating numbers with points in space. Here two other useful coordinate

systems are discussed,

(1) Cylindrical coordinates,

(2) Spherical coordinates.

Cylindrical Coordinates :

Let P be a point in space, and suppose that its cartesian coordinates are (x,y,z). Let r and Θ be polar coordinates of the point $(x,y,0)$ in the XY-plane (see Figure 2.9), then we say that (r,Θ,z) are *cylindrical coordinates* of P . From section 2-2.9.2, we know that $x = r\cos\Theta$ and $y = r\sin\Theta$. Thus the cartesian coordinates of P are related to the cylindrical coordinates of P by the equations,

$$x = r\cos\Theta, \quad y = r\sin\Theta, \quad z = z.$$

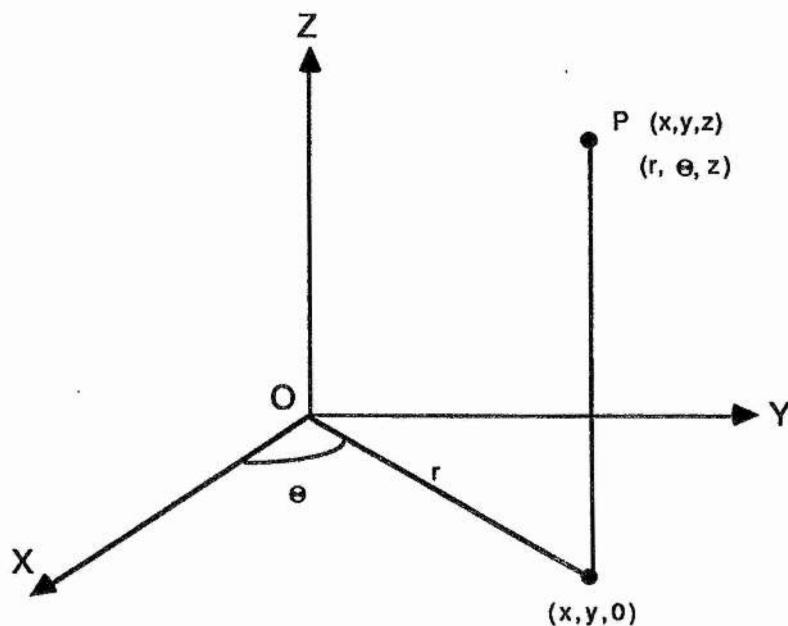


Figure 2.9

A coordinate surface in a given coordinate system is a surface that is obtained by "fixing" one of the coordinates. For example, in a cartesian coordinate system the coordinate surface $\{(x,y,z) \mid y = 2\}$ is a plane. In the case of a cylindrical coordinate system there are two kinds of coordinate surfaces. The coordinate surface $\{(r,\Theta,z) \mid z = a\}$ is a plane that is parallel to the XY- plane, and the coordinate surface $\{(r,\Theta,z) \mid \Theta = b\}$ is a plane that contains the Z axis. But the coordinate surface $\{(r,\Theta,z) \mid r = c\}$ is a right-circular cylinder; it is from this fact that the name cylindrical coordinates is derived.

Spherical Coordinates :

Again let (x,y,z) be the cartesian coordinates of a point P and let r and Θ, $r >= 0$, be polar coordinates of the point $(x,y,0)$ in the XY-plane. Suppose the angle between OP and Z axis is Φ, where $0 <= \Phi <= \Pi$, and let distance $OP = \rho$ (see Figure 2.10). Then the numbers (ρ, Θ, Φ) are called *spherical coordinates* of P. Spherical coordinates are related to cartesian coordinates of the point P as :

$$x = \rho \cos \Theta \sin \Phi,$$

$$y = \rho \sin \Theta \sin \Phi,$$

$$z = \rho \cos \Phi.$$

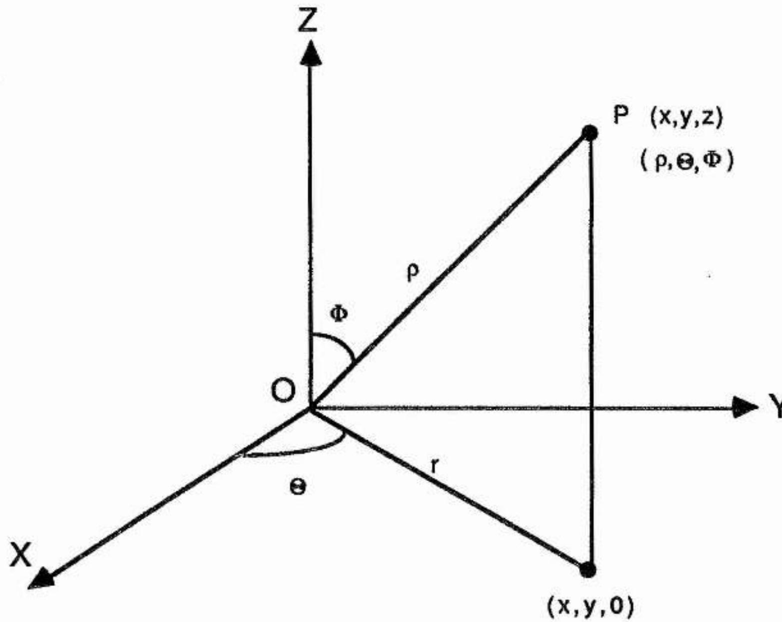


Figure 2.10

In our spherical coordinate system, the coordinate surface $\{ (\rho,\Theta,\Phi) \mid \Theta = a\}$ is a plane that contains the *Z* axis. The coordinate surface $\{ (\rho,\Theta,\Phi) \mid \Phi = b\}$ is a cone whose vertex is the origin(unless $b = 0$, $b = \Pi$ or $b = \Pi/2$, in which case the cone "degenerates"). The coordinate surface $\{(\rho,\Theta,\Phi) \mid \rho = c$, where $c > 0 \}$ is a sphere; it is from this fact the name spherical coordinates is derived.

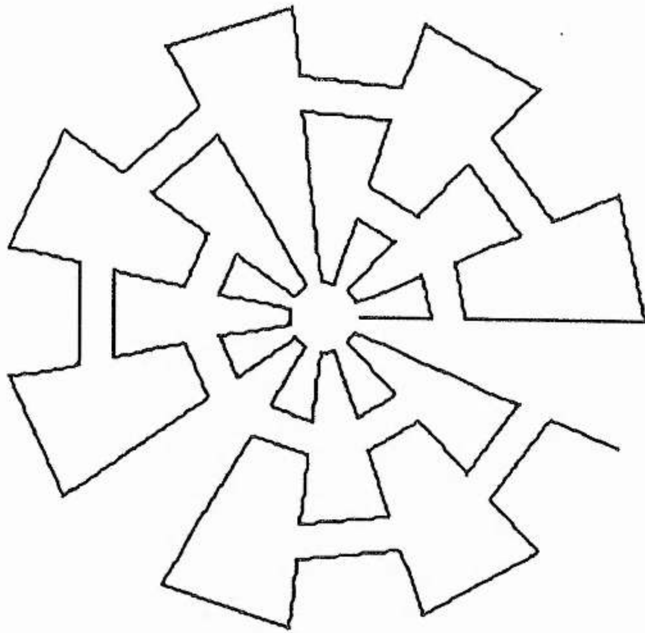
2-2.10 Implementation of Planar Polar Murray Scan :

Here we use the fast murray scan algorithm, which is given in section 2-2.6 to draw a polar murray curve. The only change which will come to the algorithm is the increment to the x and the y part. For simplicity we will replace x by r and y by Θ to get a polar coordinate (r,Θ). The number of sectors will depend upon the product of y radices and the radius of the circle will be given by the product of x radices. The angle between the two sectors will then

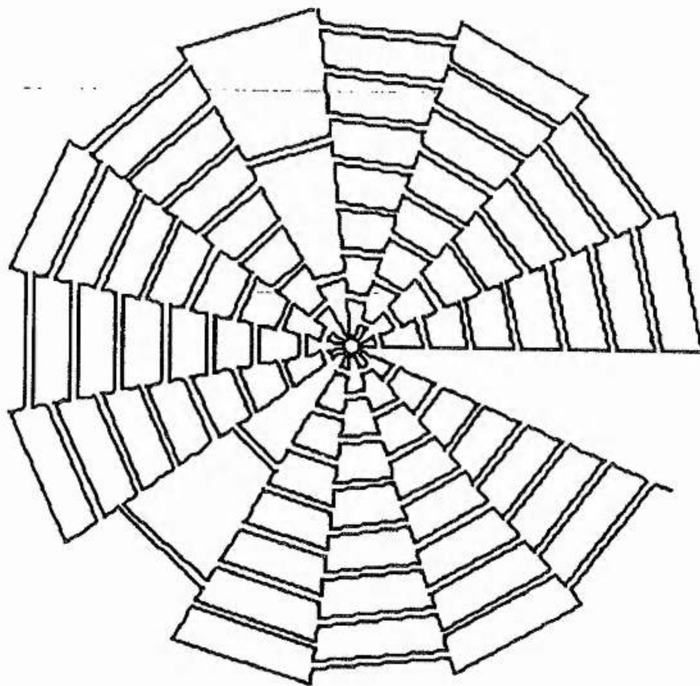
be equal to 2Π/no.of.sectors. The above two statements can now be coded as follows. The procedures used are given in section 2-2.6.

```
! The statement that determines the change in the coordinates.
! Input parameters are murray integers , radices and parities.
let digit.change = increment(digits,radices,1)
change.parity(parities,digit.change)
let inc = if parities(i+1)      then 1.0 else -1.0
if (digit.change rem 2 = 1 ) then r := r+inc
                               else Θ := Θ+inc*angle
! To plot the points we have to convert
! them into cartesian coordinates.
x := r * cos (Θ )
y := r * sin  (Θ )
```

The rest will be the same. Examples of several polar murray scan for different radices are given in Figure 2.11. This algorithm is slow due to the real arithmetic used for calculating (r,Θ). Further we have to use (r,Θ) values to find the x and the y coordinates, which is also in real arithmetic. Much of the time is wasted in calculating cos(Θ) and sin(Θ), which makes the algorithm very slow. The efficiency for the polar murray scan can be increased if we precalculate the values for cos(Θ) and sin(Θ) and put them in a array, which can be examined in the programme any number of times. Further the increment for r and Θ will be in integer arithmetic. The changed programme is given below,

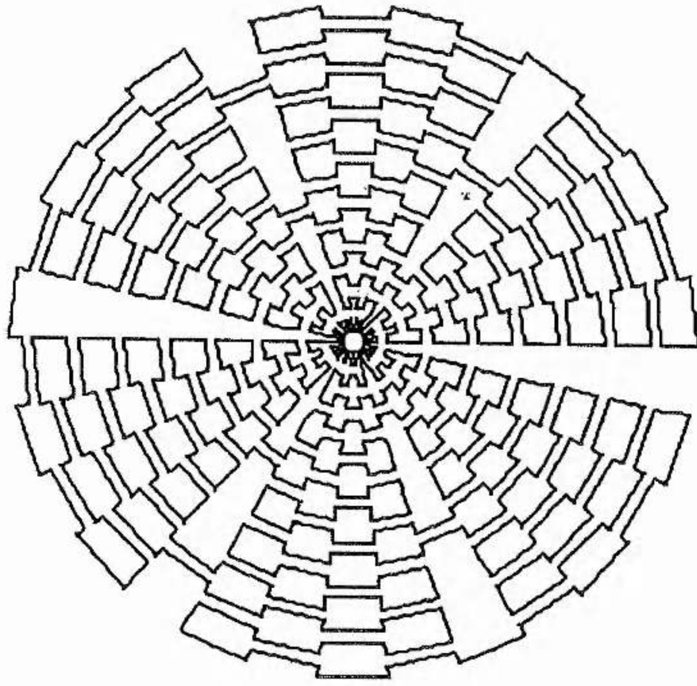


(a) Polar murray polygon with r radices 3 3 and theta radices 5 3

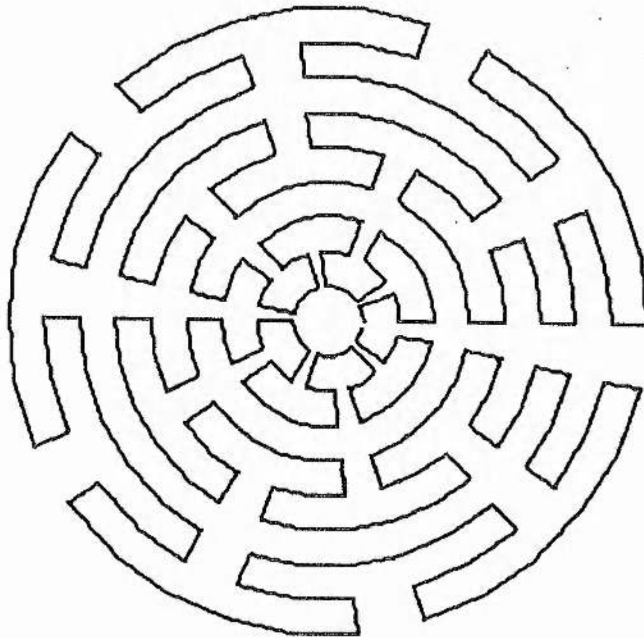


(b). Polar murray polygon with r radices 3 3 5 and theta radices 5 3 1

Figure 2.11. Polar murray polygons.

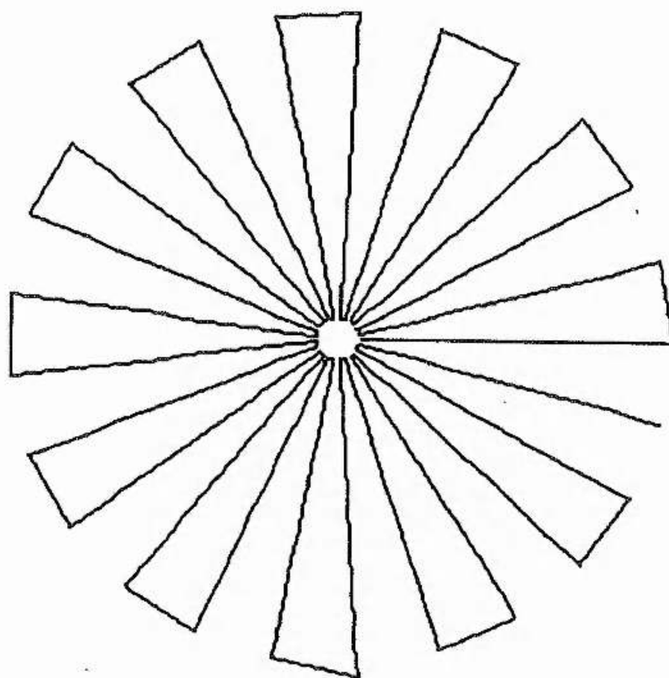


(c) Polar murray polygon with r radices 2 3 5 and theta radices 5 3 2.

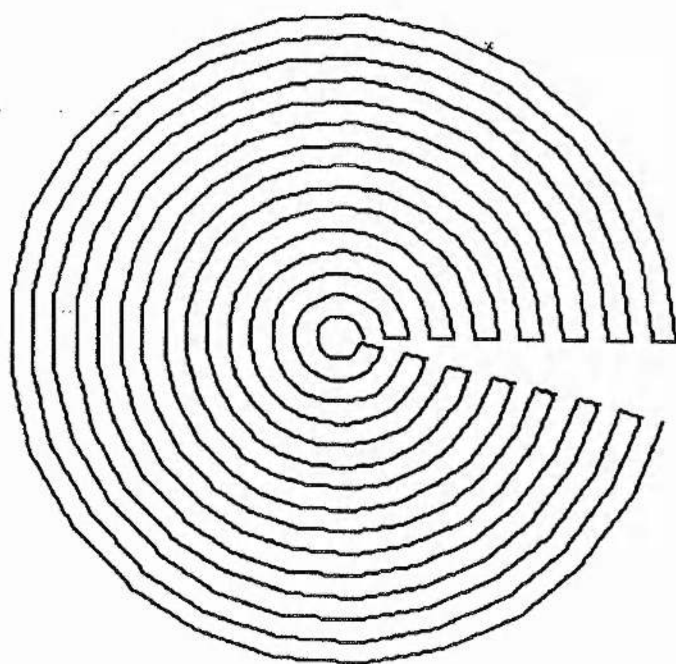


(d). Polar murray polygon with r radices 1 3 3 and theta radices 3 5

Figure 2.11(contd). Polar murray polygons.



(e). Polar murray scan with r radices 15 and theta radices 25.



(f). Polar murray polygon with r radices 1 15 and theta radices 25 1.

Figure 2.11(contd). Polar murray polygons.

Figure	radices					
	r_6	r_5	r_4	r_3	r_2	r_1
a			3	3	5	3
b	1	5	3	3	5	3
c	2	5	3	3	5	2
d	2	3	5	3	3	1
e					25	15
f		1	15		25	1

Figure 2.11 (contd). Mixed polar murray polygons.

```
!These statements will calculate the values for sine and cosine.
!The values are stored in an array.
let sine   = vector  1:: no.of.sector-1  of 0.0
let cosine = vector  1:: no.of.sector-1  of 0.0
for i = 0 to no.of.sector do
begin
        sine ( i +1 )   := sin (Θ * i )
        cosine ( i +1)  := cos ( Θ * i )
end
!This statement will give the cartesian coordinates.
if (digit.change rem 2 = 1 ) then r := r+inc
                             else Θ := Θ+inc
x := r * cosine (Θ )
y := r * sine (Θ )
```

2-2.10.1 Cylindrical Polar Murray Scan :

This is similar to the previous algorithms discussed in section 2-2.8.1.The only difference is in the coordinate systems. As defined above, the coordinate surface $\{(r,\Theta,z) \mid r = c\}$ is a right-circular cylinder. The algorithm will scan the the first plane/tile then will go to the second plane/tile and so on. The coordinate z will be kept constant for a plane.

2-2.10.2 Spherical Polar Murray Scan :

The coordinate surface $\{ (\rho,\Theta,\Phi) \mid \rho = c$, where $c > 0 \}$ is a sphere, as defined above. To get a spherical murray scan , ρ is to be kept constant. The increment for the angle between the two sectors is given by,

angle = Total.angle/product.of.radices.

For Θ the increment will be angle1(say) = 2Π/product.of y-radices, and for Φ,

angle 2(say) = Π/product.of.z-radices. The rest is the same as in the algorithm discussed in section 2-2.8.1.

2-2.11 Applications Areas :

Murray scans can be used to process an image. Many application areas of image processing are covered in this work, which are dicussed later in the following chapter. However in this issue we will discuss the scanning part using murray polygons.The major applications are also mentioned briefly.

2-2.11.1 Scanning:

A graphics screen can be considered as a finite rectangular array of pixels where each pixel is addressed by integer coordinates. A picture or image may likewise may be considered split into a finite number of cells. These arrays can be scanned in total or part by an appropriate murray polygon (or scan). A murray scan will pass through each pixel in an image recording the colour information and the number of successive pixels with the same value. It is discussed in detailed in the next chapter.

2-2.11.2 Applications

Major application areas appear to be:

1. scaling,
2. object identification,
3. operations on images using run lengths,
4. set operations,
5. hidden surface removal and shading,
6. ray tracing,
7. superimpositions of images,
8. data compaction for storage and transmission,
9. halftoning.

2-2.12 Remarks :

There are several expected advantages from a murray scan when compared to a linear scan. Firstly, as the murray scan by its nature will pass through many points close to each other it will be able to take advantage of any local correlation between pixels. This should be a considerable advantage as many images have a strong local correlation. Secondly, the murray scan will in general change direction frequently within a relatively small and compact area, thus may reduce the common patterning resulting from the more regular linear scan. Lastly, murray scans have considerable flexibility allowing change of basic tile pattern, scan order, scan direction and even dimension of scan.

Another common representation is by quadtrees. This is included as a special case of a Hilbert scan. Murray scans are also similar to quadtree encoding since the data is stored similar to quadtree 'scanning' based on the number of subdivisions of a basic tile. The only trouble with the quadtree encoding is that it can process only square images, whereas murray scans can cover a rectangular area immediately without modification.It is a major advantage over a quadtree representation. An important feature of the method is the ability, to carry out calculations and operations on the run lengths themselves without returning to the original image.

Disadvantages of the murray scan are :

1. when adjacent points in an image are a long way apart on the scan sequence. This case is also true in the case of quadtree(or octree) and other space filling curves.

2. secondly when the dimensions of an image are prime numbers. For example, x-dimension is 17 and y-dimension is 31,which implies that only one

linear murray scan can be used to scan an image; the coherence between the pixels will be lost. In this case one can increase or decrease the size of an image, to get the suitable factors. For the above example the new size for the image can be 15*33 etcetera.

Chapter 3

3. SCANNING AND DRAWING OF THE IMAGES 64

3 - 1 Introduction 64

3 - 2 Structure And List Processing 65

3 - 3 Linked List 68

3-4 Image Construction 69

3-5 Storing an Image in a Database 72

Retrieving an Image From a Database 74

3 - 6 Scanning And Drawing Of An Image 74

3 - 7 Remarks 80

3-1 Introduction :

The essence of programming requires efficient algorithms for accessing the data both in main memory and on the secondary storage devices. Further the efficiency of a programme is directly linked to the structure of the data being processed. A data structure is a way of organizing data that considers not only the items stored but also their relationship to each other. In PS-algol[Carrick, Cole, and Morrison(1987), and Morrison(1988)] any data item is allowed the full range of persistence. By persistence of data we mean the length of time that the data exits. In this language persistence is provided by an extensible number of roots known as the database. It is only necessary for a programmer to identify which data is to persist and in which database it should persist. All the images which are used in this work are initially stored in a database. Here images are the data information. The images which are stored in the database can be called anywhere in the programme and can be scanned using murray polygons. Scanning will reduce the storage space for an image by producing the sequence of runlengths with their associated colour. Once the runlengths are obtained then we can store them either in a file or in a database with a suitable data structure defined. The runlengths can be processed thereafter. They can either be used to draw the image again on the screen at any given point or to carry out any other operation.

In this chapter we will discuss briefly the data structures used for the images and for the runlengths to store them in the database, how to construct the images using PS-algol graphics facilities, and how to store and retrieve an image from the database has discussed in detail. Further how a murray scan can be used to scan the image and to draw the image back on the screen at any given point is discussed in detail. The language used is the

PS-algol[Carrick, Cole, and Morrison(1987), and Morrison(1988)] .

3-2 Structures and List Processing :

It often useful to collect together several pieces of information and give a name to this collection. For example, information about a person's name, his passport number, age, country etcetera. All this information can be held as one unit of data by declaring a data structure as follows,

structure visitors(**string** name; **int** passport.number, age; **string** country)

This defines the form of the structure and gives names to the items in the structure and also a name that is *visitors*, to this type of structure. We can now set a structure by giving information about a visitor. For example,

let A := visitors(**reads**(), **readi**(), **readi**(), **reads**())

If we want to refer about the visitor's name whose structure name is 'A', we will write

visitor's.name = A(name)

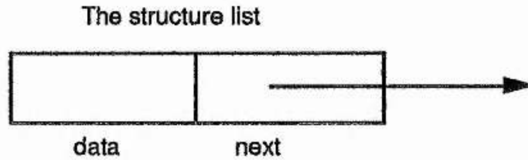
visitor's.country = A(country)

We can also define structure with pointer members that may refer to the same structure type or it may be a new structure type. For example, we can define a structure as,

structure list(**int** data; **pntr** next)

This declaration of a list can be stored in two words of memory. One word stores the member data and the second stores the member next. The pointer variable next is called a link. Each structure is linked to a succeeding

structure by the way of the member *next*. This can be displayed pictorially with links shown as arrows.



The pointer variable *next* contains an address of either the location in memory of the successor *list* element or the special value NIL which is used to denote the end of the list.Three structures each of type *list* can be defined as

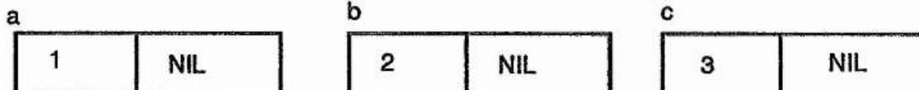
let a := *list*(1,NIL)

let b := *list*(2,NIL)

let c := *list*(3,NIL)

The result of this code is shown below,

Assignment



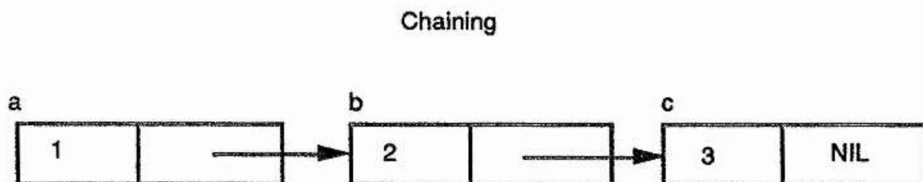
If we want to chain them together we will write,

a(next) := b

b(next) := c

These pointer assignments result in linking a to b to c, see below,

Chaining



Now these links allow us to retrieve data from successive elements. Thus,

a(next,data) := 2

and a(next,next,data) := 3

In C we don't have the facility as in PS-algol. In C a structure with a pointer member points at the same structure type. If we want to point at a different structure type then we have to redefine its type. For example two structures in C and PS-algol are as follows,

C structures	***PS-algol structures***

struct list {
    int data;
    struc list *next;
    }
	structure list(**int** data; **pntr** next)

struct points{
    int x, y;
    struc points *right;
    }
	structure points(**int** x, y; **pntr** right)

Both the structures have different types. If we want to link the second structure to the first structure then in PS-algol,we will simply write,

```
list(next) := points(x,y, nil)
```

In the case of C we first define the type of that structure which we want to link i.e.,

```
list -> next = ( struct list ) points ;
```

which is not troublesome in the case of two or three structures. But if we are dealing with many structures then it can be very cumbersome, since each time you have to redefine its type.

3-3 Linked Lists :

A structure involved in many data processing activities is the ordered list of data elements. A ordered list of three integers is shown in section 3-2. Such a data set can be represented by one dimensional array in which the jth subscript corresponds to the jth item in the ordered list. It has a head pointer addressing the first element of the list and each element points at a successor element. In the last element the link value is NIL. Such a list is referred as *linked-list.*

A list can also contain more than one pointer. A list with two pointers i.e. next and left, is a *doubly-linked-list.* The next link is a pointer to the next node in the list, whereas the left link points to the preceding node. If the left pointer or the next pointer is NIL, it indicates the end of the list. Once a list has been formed, further processing can be done onto it. For example we may have to add one more item to that or to delete one item out of it.

3-4 Image Construction :

The PS-algol graphics facilities provide a method of manipulating images for bitmapped displays integrated with a line drawing systems. Line drawings have the data type **picture** written as **pic** and bitmaps have the data type **image** written as **#pixel.**

An image is a 3-dimensional object made up of a rectangular grid of pixels. A pixel has a depth to reflect the number of planes in the image and an image has an X and Y dimension to reflect its size. In its most degenerate form a pixel is one spot which is either **on** or **off** . For example,

let a.pixel = **off**

creates a pixel a.pixel with depth 1. If we want a pixel with depth 4 we may write,

let a.pixel = **off** & **on** & **off** & **on**

which creates a.pixel with depth 4. The simplest way of constructing an image with an X and Y dimension different from 1 can be achieved by writing

let an.image = **image** 5 **by** 10 **of on**

which creates an.image with 5 pixels in the x-direction and 10 pixels in the y-direction all initially on and of depth 1. The origin of all images is (0,0).

Another way of construction of an image in the 2D case is by the picture drawing facilities of PS-algol, which allows the user to produce line drawings in an infinite two dimensional real space. Pictures may be mapped onto an image. Once a picture has been mapped onto an image it may be manipulated as an image or drawn as an image. A picture are usually built up of a number of sub pictures. The simplest picture is a point. For example,

let point := [5.3 , 2.6] declares a picture with name point. Pictures may be joined together using the join operator '^'. For example,

let square = [0,0] ^ [2,0] ^ [2,2] ^ [0,2] draws a picture, see Figure 3.1(a).

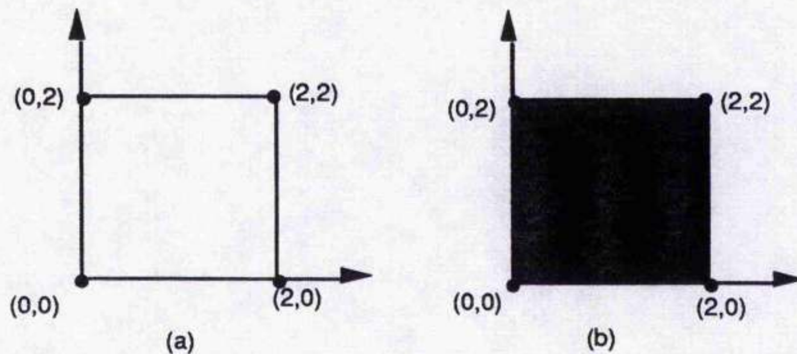


Figure 3.1

To get an image we can map a picture onto an image. For example,

draw(**screen**,square,-x,x,-y,y)

will draw the section of the picture square which is bounded by -x,x,-y,y in its coordinate space. The standard identifier **'screen'** is an image. A picture which has been mapped onto an image can be filled by any colour specified by using the standard function **fill.** For example

fill(square, **off**,1,1)

will fill the square with black, see Figure 3.1(b). In this work most of the images are constructed using this techniques.

In case of 3-dimension an image can be written as,

let 3D.image = **image** 5 **by** 20 **of on** & **on** & **off** & **off** & **on**

which creates an image 3D.image with 5 pixels in the x-direction, 20 pixels in the y-direction and having depth equal to 5 (i.e., 5 planes). The planes of the

pixel are numbered from zero . We can also use many planes to build a three-dimensional image; by simply putting them one after the other. Different three-dimensional images can be obtained by simply interchanging the position of the planes. In the case of the Sun 3/60 the maximum depth for a three-dimensional image is equal to 24 i.e. 8 bit planes per colour. The depth of an image can further be increased by joining two or more three-dimensional images each of depth 24.

Another way of constructing an image is by using the mathematical formula given for an object. Using a formula we can compute the points which will give us a picture and to get an image we can map them on an image (e.g., **screen**).

All the images can be interrogated by a standard function **Pixel.** For example,

Pixel(*an.image*, 2,3)

will return the pixel value at position 2,3 of the image *an.image*. For black and white images this value can be **on** and **off** only. In case of three-dimensional images each plane can be accessed separately by using the command given as,

let b = 3D.image(0, 1)

start from 0th plane   consider one plane

which ask the program to start from 0th plane and consider one plane only. Here it yields a plane which is the first plane of the 3D.image.

All two dimensional images which are used in this work are given in Figure 3.2. Some of the three dimensional images are obtained using these two dimensional images by putting them one after the other.

3-5 Storing an Image in a Database :

All the images which are used in this work are initially stored in the database . To show, how images can be generated by using line drawing facilities and how images can be stored and retrieved from the persistence store an example is given of a program. In this example, it is assumed that the database root is a pointer to a data structure for associative storage and retrieval, supported by PS-algol, called a table. Entries are placed in the table using the procedure *s.enter* which takes the associative key, the table, and the value to be stored. The procedure s.lookup retrieves a value from the given table using the given key

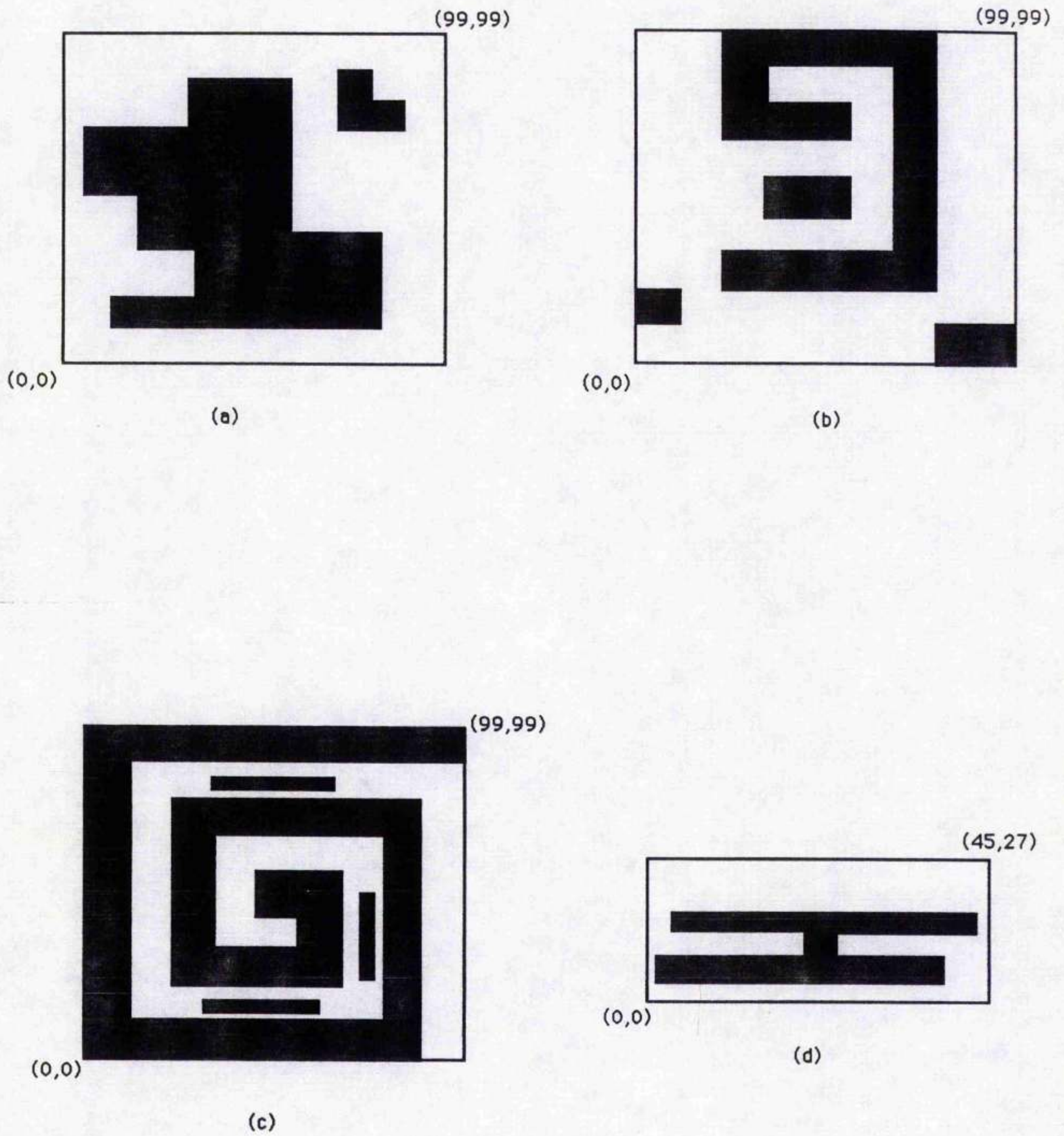
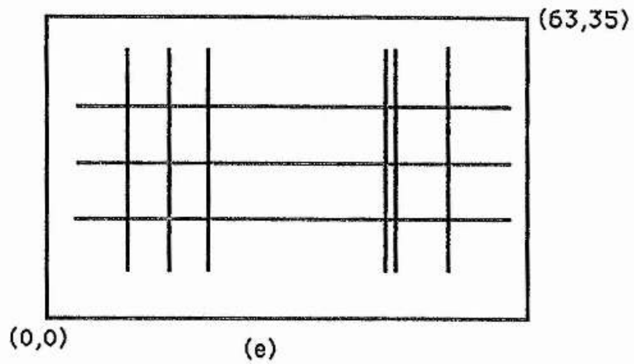


Figure 3.2. Two-dimensional images which are obtained by using the line draw facilities of Ps-algol.



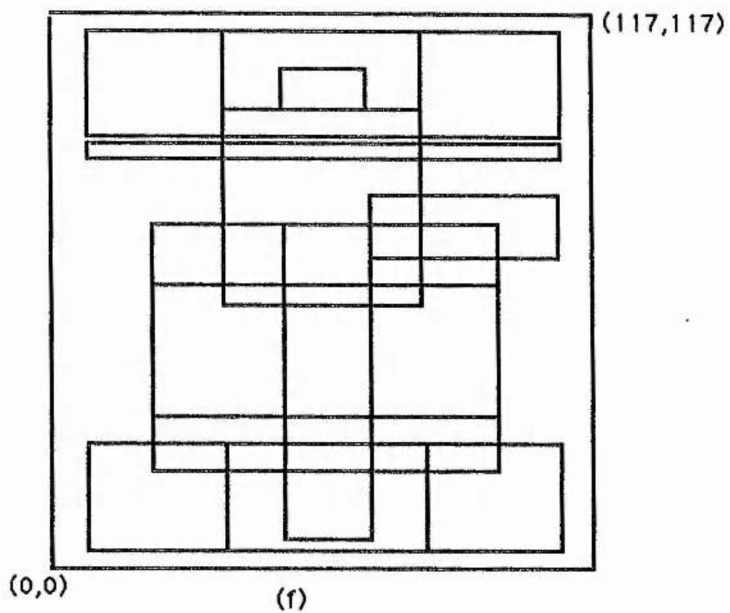


Figure 3.2 [contd] . Two-dimensional images

Note : The rectangle enclosing an image is not included in the ima

```
! Structure used to store the image.
structure image.container(c#pixel an.image)
let db = create.database("an image","pass")
if db is error.record do
begin
  write "unable to create database :",db(error.explain),"'n"
  abort
end

! This block will make a small window on the screen
let X = X.dim(screen) div 2 ; let Y = Y.dim(screen) div 2
let Image = limit screen to 100 by 100 at X,Y

! This will draw the picture using line drawing facilities.
let a = [30,20]^[80,20]^[80,50]^[50,50]^[50,80]^[40,80]^
            [40,70]^[10,70]^[10,60]^[20,60]^[20,40]^
            [40,40]^[40,30]^[30,30]^[30,20]
let b = [70,70]^[95,70]^[95,80]^[90,80]^[90,90]^[70,90]^[70,70]
let a.pic = a & b

! To get an image we have to map a picture on an image.
draw(Image,a.pic,0,100,0,100)
fill(Image,on,50,40)          ! This will fill the area with value on
fill(Image,on,80,80)

! a structure containing an image Image
! associated with the key "Image"
s.enter("Image",db,image.container(Image))
if commit() = nil do write "the Image entered in the  database 'n"
```

Program 3: A program to store an image in a database

The database called "an image" now contains a table with a key "Image" which has an associated value of a structure that contains the description of the image. This is shown in Figure 3-3.

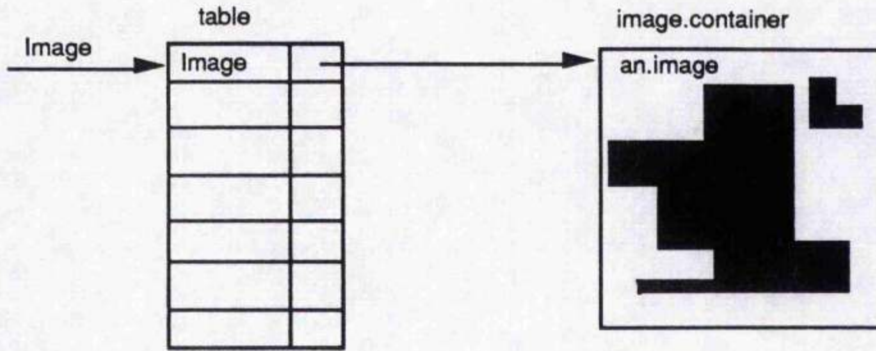


Figure 3.3. Pictorial representation of the database "an image".

3-5.1 Retrieving an Image from a Database:

The next example retrieves an image description from the database and places it on the screen for further processing on it.

```
! Structure used to store the image.
structure image.container(c#pixel an.image)
let db = open.database("an image","pass", "read")
if db is error.record do
begin
   write "unable to create database :",db(error.explain),"'n"
   abort
end

let get.image = s.lookup("Image",db)(an.image)
copy get.image onto limit screen at 100,100
```

Program 4: A simple program to retrieving an image from the database.

3-6 Scanning and Drawing of an Image:

As defined earlier an image is nothing but the collection of pixels or dots defining a rectangle. Since a murray scan is a space filling curve, it will pass through each and every pixel in an image recording the colour information and the number of successive pixels with the same colour. Before scanning an

image we have to decide the murray radices. Murray radices can be obtained by factorising the x and the y dimensions of an image. Only point to remember is that the product of the x-radices must be equal to the x-dimension of the image and similarly for the y-radices. If the dimensions of an image cannot be factorised then we will use only two murray radices equal to the x and the y-dimensions of an image. The sequence of colour and run length are then coded to minimise the data required to describe the image. The image is usually split into bit planes and each bit plane considered separately as a black and white image (i.e. *on* or *off)*. In the case of black and white images there is no need for colour information as the run lengths alternate between black and white. However, the first output in the sequence must obey a convention, usually taken as first run length is white. If the first run length is one of black pixels then a zero is output first. With this information an image can be easily described and thus reconstructed from the run lengths at any required position. For example, the simple rectangular image (Figure 3.4), have been scanned using a murray polygon with x-radix 6 and y-radix 3.

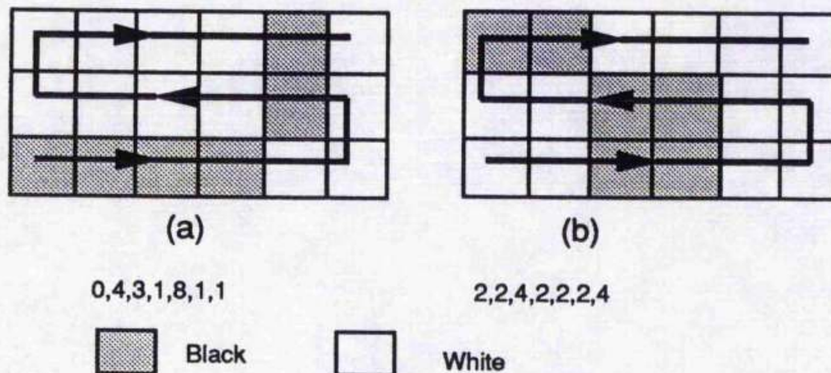


Figure 3.4. (a) Here the first runlength is zero since the starting point in the image belongs to the black cell.

(b) Here the first runlength is not zero since the starting point in the imagebelongs to the black cell.

The collection of runlengths and the murray radices used to scan the image are then stored in the database or in a file and can be used for further processing which we will discuss in the following chapters.

It is assumed that the murray scan will be able to take advantage of any inherent structure in the image. Since the murray scan is moving around in two dimensions a point on the murray scan will have four possible directions to move rather than the standard linear scan with fly back which has only one possible direction to move. The pixels coherence can therefore be exploited in the case of the murray scan than that of standard linear scan with fly back. Hence a murray scan with its localized scanning patterns has a better chance of capturing a few long runs of pixels of the same value. One could conclude that the murray scan will therefore produce less runlengths than that of the standard linear scan with fly back. But it is not always true. The explanation for this is as follows. Consider a large homogeneous connected color blob with a well defined boundary as shown in Figure 3.5.

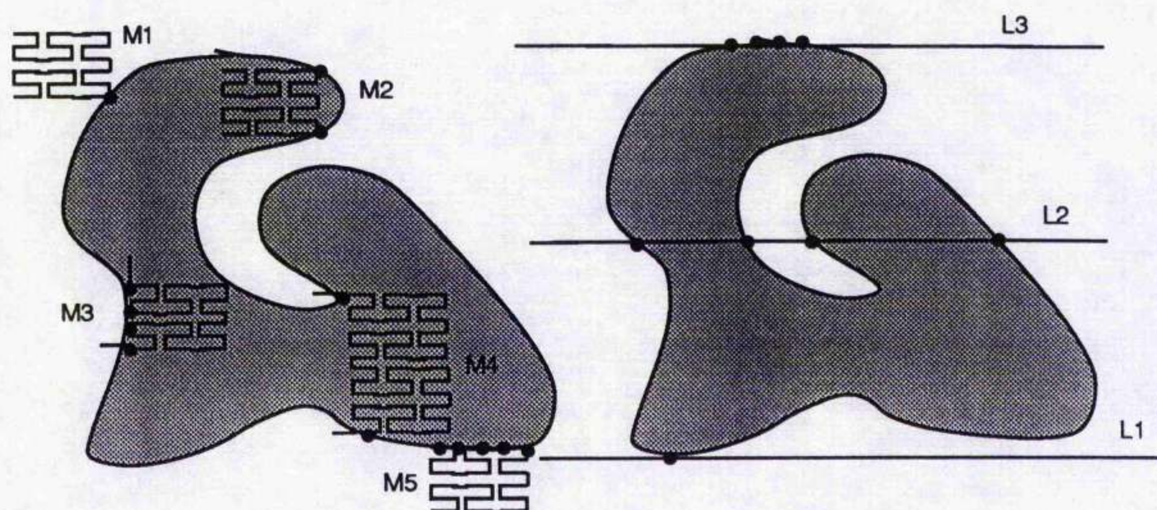


Figure 3.5. Boundary point possibilities in the two cases.
a) Murray scan. In case M1 the scan touches the boundary only in one point, whereas in the case of M2 - M5 , it touches the boundary either in two points or more.
b) Linear scan . In case L1 it touches the boundary in only one point i.e. tangent. In the case of L2 and L3 the boundary points are two or more.

Most linear scan run lengths will pass through two of these boundary points for each run corresponding to the entry and exit points of run. There will be exceptions to this case when a horizontal line touches only one boundary point and similarly when a part of the boundary is itself a horizontal line. The exceptions tend to cancel each other and the number of run lengths is roughly proportional to half the number of boundary points on the blob. In the case of a murray scan these exceptions will be different. Since a murray scan is frequently changing direction there are likely to be more instances when the scan either meets the color blob in just a point or several boundary points and other cases when an internal run meets the boundary and then turns back into the blob thus giving a long run length associated with a particular blob (see Figure 3-5a). Since boundary points are used up in a different way to that of a standard linear scan, the total number of runlength will be affected. Further in the case of a standard linear scan a break of runlength can be a minor advantage in favour of murray scans. Since a murray runlength passing through an interior of a color blob will be long, there will therefore be some long

runlength associated with a particular color blob and consequently since the total number of runs is approximately constant in comparison to linear scan with flyback, there will be a large number of short run lengths to compensate for this and thus will have a different distribution to that of a standard linear scan. This distribution may be exploited in a final coding of the run length for storage or transmission. For more detail refer Buntin(1988).

3-6.1 How to draw images using a sequence of runlengths :

Since the runlength sequences are associated with their color, the reconstruction from the runlengths at any required position will be very simple. We will explain this with the help of an example. The color information, runlengths, and the murray radices corresponding to an image is given below,

color	-> w b w b w b w
runlengths	-> 4, 5, 1, 1, 2, 1, 1
x-radices	-> 3
y-radices	-> 5

We will start from the first point i.e., (0,0). Since the first runlength is 4 and is white, we will move four steps in accordance to the scan direction, giving value **on**(or 1) to these four points. Next runlength is 5 and it corresponds to the black cell, so the next 5 points will get the value **off**(or 0) and so on. The final image is given in Figure 3.6

Figure 3.6. Construction of an image from a sequence of runlengths.

The procedure which will draw the image back on the screen using the runlengths is given below. This procedure will take a linked list of runlengths and the murray radices.

```
! The input are the murray radices and the list of runlengths.
let draw.image = proc(*int r; pntr LIST)
    begin
        let digits     = vector 1 :: upb(r)+1 of 0
        let  parities  = vector 1 :: upb(r)+1 of true
        ! X and Y can be of any size
        let window  =  limit screen to X by Y at 100,100
        let x := 0
        let y := 0
        while LIST ~= nil do
        begin
            let pixels := LIST(run.length)
            let white := if LIST(col)="w" then off else on
            let B.or.W.square := image 1 by 1 of white
            for i = 1 to pixels do
            begin
                if LIST(next) ~= nil or (i < pixels) do
                begin
                        copy B.or.W.square onto limit window at x,y
                        let i  =  increment(digits,r,1)
                        change.parity(parities,i)
                        let inc := if parities(i+1) then 1 else -1
```

```
                    if i rem 2 = 1 then x:= x +inc  else y := y +inc
                end
            end
            LIST:=LIST(next)
        end
    end
```

3-7 Remarks :

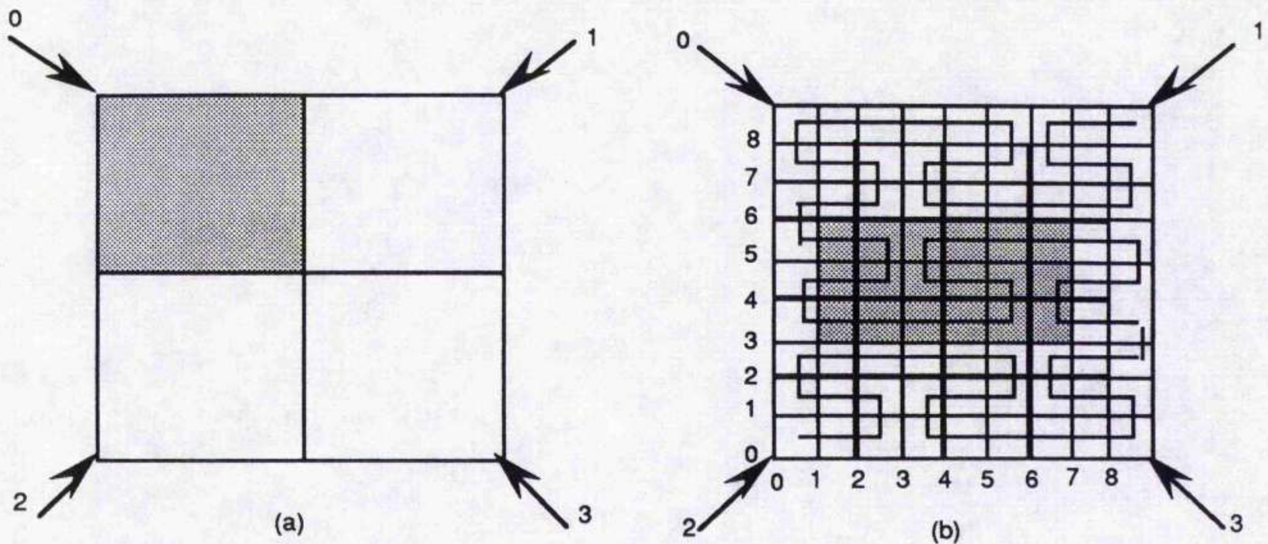
In comparison to the standard linear scan with fly back a murray scan will be slow, since it requires more calculations per step than that of a standard linear scan. However, hardware can be built to compensate for this. However a murray encoding will in general be more compact than that of standard linear scan (as discussed in section 3-6) .

In comparison to the murray approach the quadtree approach may take more time to scan an image. This is due to the extra preprocessing steps used in obtaining the quadtree. To form a quadtree, the very first step which is required is to convert the rasters (i.e., runlengths) into a quadtree (refer Samet(1981)). The scanning is generally done by a standard linear scan.The next step is to traverse the quadtree to merge groups of four pixels or four blocks of a uniform color. In the case of a linear quadtree we have to apply condensation and sorting to the collection of codes obtained after transforming rasters (i.e., runlengths) into a linear quadtree (refer Unnikrishnan and Venkatesh(1984)). In the case of the murray approach, we do not need to process the runlengths, once obtained after scanning an image. Hence a murray approach may be less time consuming than that of forming a quadtree or linear quadtree.

In order to obtain the compacted codes for an image, both approaches (i.e., murray and linear quadtree), will be equally effective, depending upon the shape of an image (*Note : we use a linear quadtree for comparison because it*

stores only black pixels rather than that of a general quadtree). The reason is as follows. In the case of quadtrees an image is divided into four quadrants if it is not homogeneous (i.e. not of the same color). A quadrant is subdivided into four subquadrants if it is not homogeneous and so on. These quadrants are then dealt separately to encode the present black pixels. In the case of murray polygons also an image is divided into small tiles, but these tiles are not considered separately as in the case of quadtrees *(Note : Tiles corresponding to the quadrant).* To get the runlengths we proceed from tile to tile gathering the pixels of the same color. Since in the case of murray polygons we deal with the whole image rather than the quadrants, hence there may be a better chance of capturing more pixels of the same color. The main advantage of linear quadtrees is to store only the black pixels, whereas in the murray approach we have to store both the black and the white pixels. Hence in some cases linear quadtrees may be more compressive than that of murray approach. But in the case of linear quadtree, smaller the black homogeneous quadrant bigger will be the code length and in the case of the murray approach, smaller the black homogeneous area smaller will be the code length. Hence nothing can be stated positively about the two approaches. Best and worst cases are always there. Here consider two cases (i.e. best and worst) to justify it.

(Note : Here the best case is in favour of the quadtree approach . We can consider similarly the best case for the murray approach also)



Note : In the case (b) , for the murray approach we have increased the dimensions by one unit .

Figure 3.7. Two cases (a) best case and (b) worst case

In the best case (see Figure 3.7a), a linear quadtree has only one code to store whereas a murray approach has at least four codes (i.e. runlength) to store the information about the image.

In the worst case (see Figure 3.7b) a linear quadtree has 12 codes to store, which are given below,

310, 301, 300, 211, 210, 201, 132, 130, 023, 021, 12, 03

whereas in the case of murray approach we required only 7 runlengths to store the image.The runlengths are,

29, 2, 4, 12, 2, 4, 29

In the case of linear quadtree only the black pixels are stored, whereas in the case of the murray method white as well as black pixels are stored .
Similarly we can also consider best and worst cases for the murray approach.

Chapter 4

4. **SCAN CONVERSION AND SCALING OF IMAGES** 83
4 - 1 Introduction 83
4 - 2 Scan Conversion 84
Method 1 84
Some Lemmas 89
Method 2 96
Some Lemmas 98
4 - 3 Implementation of Scan Conversion 104
Data Structure 104
Scanning 107
Algorithms 108
Comparison Between The Two Algorithms 112
Comparison Between Linear and General Murray Scan 113
4 - 4 Scaling 114
Introduction 114
Scaling Using Murray Polygons And Its Implementation 116
Some Lemmas 122
Theorem 123
Results 132
4 - 5 Remarks 134

4-1 Introduction :

As discussed in chapter 1, murray polygons can be used to scan rectangles of different sizes. A rectangle can be scanned either by using horizontal murray scans or by using vertical murray scans. By a horizontal murray scan we mean that the displacement in the y-direction will only be of one unit, either increment or decrement and vice-versa. As discussed earlier, the basic direction of the scan from horizontal to vertical can be changed by making the least significant radix take value 1. Further a vertical scan can be obtained by changing the positions for the radices and the values, corresponding to the x-part and the y-part, more detail follows.

An image represented by an n*n array of pixels would need too much space to store it in uncoded form. The exact data compression can be achieved by runlength encoding. By exact data compression we mean, to restore the same image, without any distortion from the collection of runlengths.The runlength sequences and their associated colour information are produced by scanning an image with a murray scan.The murray scan will pass through each and every pixel recording the colour information and the number of consecutive pixels of that colour. The data can further be compressed by coding the runlengths. We can compress the data either to give the exact compression or to give an approximate compression. By approximate data compression we mean, once the data has been compressed we cannot restore the same image from the collection of runlengths. Here some of the information is going to be lost.The runlengths can be scaled up or down as required. The only point to remember is that, if we have to scale the image in the x-direction then we will use a horizontal murray scan to scan the image and for the y-direction scaling we will use a vertical murray scan. If we want to scale an image in the x-direction as well as in the y-direction the simplest

way of doing it is to scale the image horizontally/vertically then draw the image on the screen using the compressed runlengths and then scan this scaled image vertically/horizontally to get the sequence of runlengths, which can be scaled in the vertical/horizontal direction. Drawing and scanning part of the image will be time consuming, so it would be better if we only work on the runlengths without going back to the image, to get the runlengths for the another scan(i.e. vertical or horizontal). We discuss how this can be done.

In this chapter the conversion from horizontal murray scan to vertical murray scan, or vertical murray scan to horizontal murray scan, is described. We call this process scan conversion. In this chapter we will refer to horizontal murray scans as scan1 and to vertical murray scans as scan2. Scaling the images either in one direction (i.e. x or y-direction) or in both directions is described. All the algorithms derived use only runlengths. Finally the results are compared for the different images shown in chapter 3. The language used for the algorithms is the Outline System of PS-algol[Carrick, Cole, and Morrison(1987), and Morrison(1988)] and C[Kernighan, and Ritchie(1978), and Kelley, and Pohl(1984)].

4-2 Scan Conversion :

Here two methods are discussed for the conversion of scan1 into scan2 and vice-versa. Both the methods use the murray run length encoding and murray radices as input.The efficiency of both methods is compared for different images.The result obtained are shown in the following sections.

4-2.1 Method 1 :

Before we describe this method, we will review some of the definitions previously defined. As shown earlier murray scans can be forced to go either in the x-direction or in the y-direction by incrementing or

decrementing the x-part or the y-part by one unit. This can be obtained simply by using a radix value of one. Another way of achieving this is by changing the x-part and the y-part. For example let r_n, r_{n-1} be the radices and d_n ,d_{n-1} be the corresponding digits.The radix r_{n-1} forces the scan to move r_{n-1} steps in the x-direction whereas the next radix r_n force the scan to repeat the previous step r_n times in the y-direction.If we interchange the radices i.e. R_n, R_{n-1} where, $R_n = r_{n-1}$ and $R_{n-1} = r_n$ and the digits also (i.e. $D_n = d_{n-1}$ and $D_{n-1} = d_n$), then the scan will go firstly in the y-direction then in the x-direction, since all odd digits now corresponds to the y-part and all even digits corresponds to the x-part. This is discussed below.

Mathematically :

Let N be the nth point on a given scan (say scan1) and,

let $d = d_n, d_{n-1}, \ldots\ldots, d_1$ be the equivalent murray integer with the radices $r = r_n, r_{n-1}, \ldots\ldots, r_1$, where n = 2k and k is an integer. Now our problem is to find the corresponding mth(say) point on a second scan, i.e., scan2. Let $c = c_n, c_{n-1}, \ldots\ldots c_1$ be the Gray coded transformation of a murray integer d, where

$$c_i = d_i \quad \text{iff } \Sigma d_j \text{ is even, (for } j = i+1, \ldots\ldots, n) \quad \text{---------} \quad (A)$$

$$= r_i - 1 - d_i \quad \text{otherwise}$$

Let $A = A_n, A_{n-1}, \ldots\ldots, A_1$ be the Gray coded integer where

$$A_i = c_{i+1} \quad \text{iff } i \text{ rem } 2 \text{ is not equal to zero} \quad \text{------------} \quad (B)$$

$$= c_{i-1} \quad \text{otherwise}$$

Since we have interchanged the Gray coded integer (i.e. $A_1 = c_2$, $A_2 = c_1$, ...etc). we also have to change the radices correspondingly .

Let $R = R_n, R_{n-1}, \ldots\ldots\ldots R_1$ be the corresponding radices such that for each i ,

$0 <= A_i < R_i$ where

$R_i = r_{i+1}$ iff i rem 2 is not equal to zero -------------- (C)

$= r_{i-1}$ otherwise.

The two scans may now be defined as :

Scan 1 (or Horizontal scan) :

To get a horizontal scan we will consider our x-part and y-part to take the values,

$x' = c_{n-1}, c_{n-1}, \ldots\ldots\ldots, c_1$ and,

$y' = c_n, c_{n-2}, \ldots\ldots\ldots, c_2$

Now simply de-gray code x' and y' parts and then convert back to ordinary integers giving (x,y) as explained in chapter 2.

Scan2 (or Vertical scan) :

If we simply interchange the values of x' and y', the two parts will be given as,

$x' = A_n, A_{n-2}, \ldots\ldots\ldots, A_2$ and,

$y' = A_{n-1}, A_{n-3}, \ldots\ldots\ldots, A_1$.

Then by simply de-gray coding x' and y' parts and then converting back to ordinary integer we will get the coordinate for the verticle scan.

To get the corresponding mth point on the second scan we can de-gray code the digits A_i to give the equivalent murray integer, which can be used to determine the corresponding point in scan2.

Let $d'_n, d'_{n-1}, \ldots, d'_1$ be the equivalent murray integer where, for each i,

$$d'_i = A_i \text{ iff } \Sigma A_j \text{ is even, (where } j = i+1, \ldots, n) \quad \text{------------} \quad (D)$$

$$= R_i - 1 - A_i \quad \text{otherwise.}$$

The corresponding mth point in the second scan can now be given as,

$$m = ((\ldots\ldots((d'_n * R_{n-1} + d'_{n-1}) * R_{n-2} + \ldots\ldots + d'_2) * R_1 + d'_1.$$

(*Note : Total number of parenthesis will be equal to n-2.)*

The steps of the transformation for a given point N, which is the nth point in scan1, to the mth point in scan2 are,

1. Convert N into the equivalent murray integer with the given radices,
2. Convert this integer into the equivalent Gray coded integer,
3. Interchange the Gray coded integer,
4. Convert back to equivalent murray integer,
5. Convert back to the mth point on scan2.

Further if a coordinate for a pixel on an image is given, the corresponding nth and mth points on scan1 and scan2 can be determined. Let (x,y) be the coordinate representing a pixel on an image. The first step to get the nth point in scan1 is to convert the x and the y coordinates to the equivalent gray coded integer as explained in chapter 2. Let the two parts be,

$$(x , y) = (c_{n-1}c_{n-3}.........c_1 , c_nc_{n-2}...........c_2)$$

Two get the mth point in scan2 we have only to interchange the x-part and the y-part and the radices and the rest will be the same as for scan1. The two parts for x and y will then be,

$$(x , y) = (c'_nc'_{n-2}...........c'_2 , c'_{n-1}c'_{n-3}.........c'_1) \text{ where,}$$

$$c'_i = c_{i+1} \text{ iff } i \text{ rem } 2 \text{ is not equal to zero}$$

$$= c_{i-1} \text{ otherwise}$$

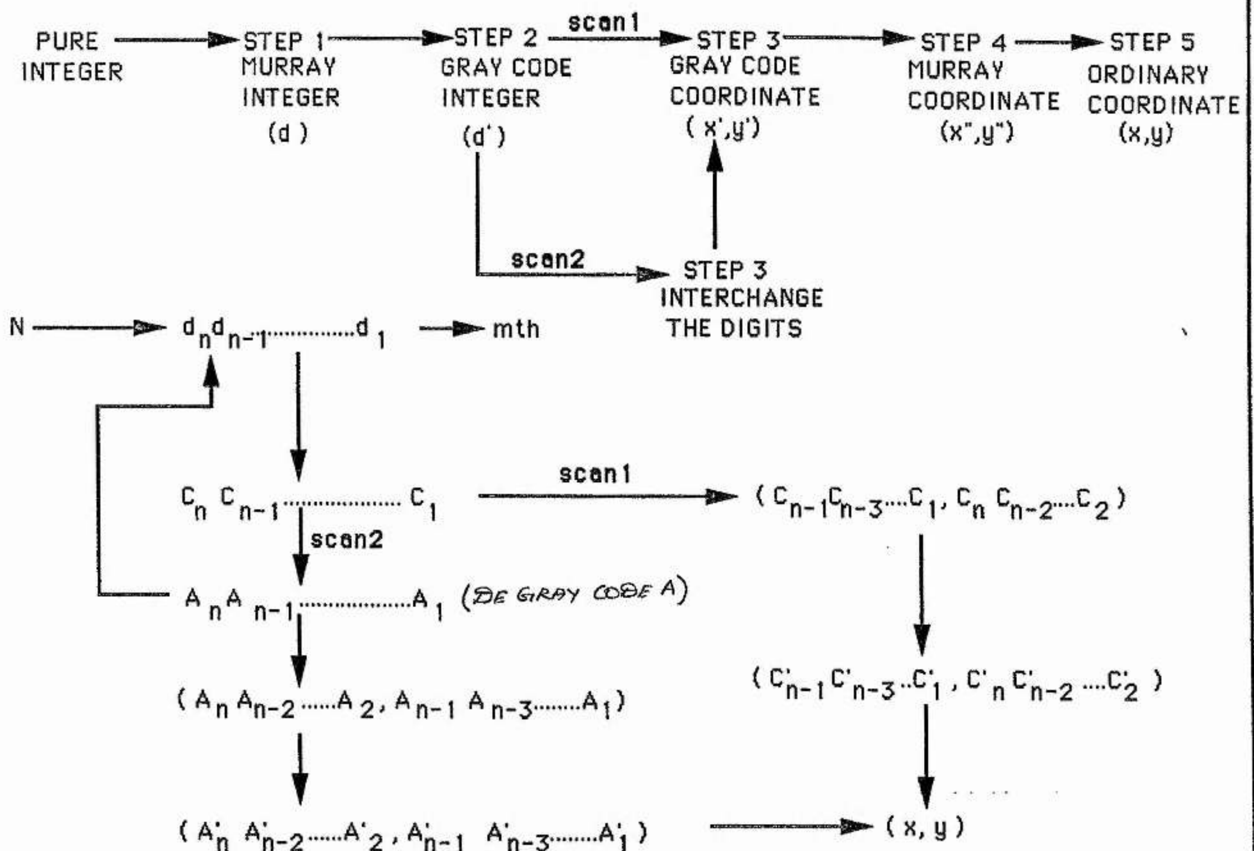
The whole scheme is given in Figure 4.1. The steps of the transformation from the murray digits of scan1 to the corresponding murray digits of scan 2 are given in Table 4.1.

Nth point	scan1 r -> 3 5 d -> d_2d_1 d	Gray-code Integer C	Interchange Gray-code Integer A	scan2 r -> 5 3 d -> d_2d_1 de-gray code A	Mth Point
2	02	02	20	20	6
9	14	10	01	01	1
13	23	23	32	30	9
14	24	24	42	42	14

Nth point	scan1 r-> 1 5 3 5 3 3 d	Gray-code Integer C	Interchange Gray-code Integer A	scan2 r-> 5 1 5 3 3 3 de-gray code A	Mth Point
11	000102	000120	001002	001220	51
435	041310	041110	401101	401101	577

Table 4.1. Transformation of murray digits of scan1 to murray digits of scan2.

Figure 4.1. Murray Transformation from N to (x,y) .



Where

d_i = N rem r_i the new value of N will be equal to (N div r_i)
Here i will start from 1 and will range upto n

$C_i = d_i$ if $\sum_{i=1}^{n} d_i$ is even,

$r_i - 1 - d_i$ otherwise.

$A_i = C_{i+1}$ iff i is odd And $R_i = r_{i+1}$ iff i is odd

C_{i-1} otherwise r_{i-1} otherwise

Scan1 :

$C'_i = C_i$ if $C_{i+2} + C_{i+4} + \ldots + \ldots$ is even

$r_i - 1 - C_i$ otherwise

$x = ((\ldots(C'_{n-1} * r_{n-3} + C'_{n-3}) * r_{n-5} + C'_{n-5}) * r_{n-7} + \ldots + C'_1)$

$y = ((\ldots(C'_n * r_{n-2} + C'_{n-2}) * r_{n-4} + C'_{n-4}) * r_{n-6} + \ldots + C'_2)$

Scan2 :

To get scan2 , replace C by A, C' by A' and r by R in the above expressions. The value for the x and y part will be given as,

$x = ((\ldots(A'_n * R_{n-2} + A'_{n-2}) * R_{n-4} + A'_{n-4}) * R_{n-6} + \ldots + A'_2)$

$y = ((\ldots(A'_{n-1} * R_{n-3} + A'_{n-3}) * R_{n-5} + A'_{n-5}) * R_{n-7} + \ldots + A'_1)$

Now for a faster algorithm we will define a few lemmas :

Lemma 1 :

Suppose d is a murray integer, c is the Gray coded integer, A is the equivalent Gray coded integer obtained from c as defined above, d' is the equivalent murray integer obtained from A, then for all odd i, we have,

$$d'_i = d_{i+1} \text{ iff } d(i) \text{ is even}$$

$$= r_{i+1} - d_{i+1} - 1 \quad \text{otherwise}$$

or

$$d'_i = d_{i+1} \text{ iff } \sum_{i}^{n} r_i - r_{i+1} - d_i \text{ is odd}$$

$$= r_{i+1} - d_{i+1} - 1 \quad \text{otherwise}$$

where n = no. of digits

Proof.

Since i is odd, then from the above equations i.e. B and C, we have

$$A_i = c_{i+1} \text{ and}$$

$$R_i = r_{i+1} \qquad (1)$$

Further from equation D we have

$$d'_i = A_i \text{ iff } \sum_{i+1}^{n} A_i \text{ is even}$$

$$= R_i - 1 - A_i \quad \text{otherwise.}$$

Using condition (1) in the above equation we will get,

$$d'_i = c_{i+1} \text{ iff } \sum_{i=1}^{n-1} c_{i+1} - c_{i+1} \text{ is even} \qquad (2)$$

$$= r_{i+1} - c_{i+1} - 1 \quad \text{otherwise.}$$

Also by using equation (A) we can get ,

$$c_{i+1} = d_{i+1} \quad \text{iff } \sum_{i+2}^{n} d_i \text{ is even} \qquad (3)$$

$$= r_{i+1} - 1 - d_{i+1} \quad \text{otherwise}$$

Now we will use equation (2) and (3) to find a equation between d and d'. We will consider four cases.

Case 1 : $\sum_{i+2}^{n} d_i$ is even and $\sum_{i-1}^{n-1} c_{i+1} - c_{i+1}$ is also even

From equation (2) and (3) we have,

$$d'_i = c_{i+1} \quad \text{and} \quad c_{i+1} = d_{i+1}$$

This implies $d'_i = d_{i+1}$.

The above two conditions (i.e., $\sum_{i+2}^{n} d_i$ is even and $\sum_{i-1}^{n-1} c_{i+1} - c_{i+1}$ is also even) can be joined to form a single condition . let us consider,

$$\sum_{i-1}^{n-1} c_{i+1} - c_{i+1} = \text{even}$$

or $$\sum_{i-1}^{n-1} d_{i+1} - d_{i+1} = \text{even}$$

$$d_i + d_{i+1} + d_{i+2} + \dots\dots + d_n - d_{i+1} = \text{even}$$

$$d_i + d_{i+2} + \dots\dots + d_n = \text{even}$$

But $\sum_{i+2}^{n} d_i$ is even, this implies, d_i + even = even

i.e., d_i = even

Case 2 : $\sum_{i+2}^{n} d_i$ is even and $\sum_{i-1}^{n-1} c_{i+1} - c_{i+1}$ is odd

From equation (2) and (3) we have,

$$d'_i = c_{i+1} \quad \text{and} \quad c_{i+1} = r_{i+1} - 1 - d_{i+1}$$

Using the above two conditions(i.e, $\sum_{i+2}^{n} d_i$ is even and $\sum_{i-1}^{n-1} c_{i+1}-c_{i+1}$ is odd) we will get,

$$d'_i = r_{i+1} - 1 - d_{i+1} \quad \text{if } d_i \text{ is odd}$$

Case 3 : $\sum_{i+2}^{n} d_i$ is odd and $\sum_{i-1}^{n-1} c_{i+1}-c_{i+1}$ is also odd

From equation (2) and (3) we have,

$$d'_i = r_{i+1} - 1 - c_{i+1} \quad \text{and} \quad c_{i+1} = r_{i+1} - 1 - d_{i+1}$$

This implies $d'_i = r_{i+1} - 1 - (r_{i+1} - 1 - d_{i+1})$

$$d'_i = d_{i+1}.$$

We can now find a single condition by joining the above two conditions (i.e, $\sum_{i+2}^{n} d_i$ is odd and $\sum_{i-1}^{n-1} c_{i+1}-c_{i+1}$ is also odd). let us consider,

$$\sum_{i-1}^{n-1} c_{i+1}-c_{i+1} = \text{odd}$$

or $$\sum_{i-1}^{n-1} (r_{i+1} - 1 - d_{i+1}) - (r_{i+1} - 1 - d_{i+i}) = \text{odd}$$

$$\sum_{i-1}^{n-1} r_{i+1} - \Sigma d_{i+1} - \Sigma 1 - r_{i+1} + 1 + d_{i+1}) = \text{odd}$$

$$r_i + r_{i+1} + \dots\dots\dots + r_n - (d_i + d_{i+1} + \dots\dots\dots + d_n) - (n-i+1) - r_{i+1} + 1 + d_{i+1} = \text{odd}$$

$$r_i + \sum_{i+2}^{n} r_i - d_i - \sum_{i+2}^{n} d_i - n + i = \text{odd}$$

$$r_i + \sum_{i+2}^{n} r_i - d_i - \text{odd} - n + i = \text{odd} \quad (\text{since } \sum_{i+2}^{n} d_i \text{ is odd})$$

$$\sum_{i}^{n} r_i - r_{i+1} - d_i - \text{even} + \text{odd} = \text{odd} + \text{odd} \quad (\text{ n is even and i is odd, given})$$

$$\sum_{i}^{n} r_i - r_{i+1} - d_i = \text{odd}$$

case 4 : $\sum_{i+2}^{n} d_i$ is odd and $\sum_{i-1}^{n-1} c_{i+1}-c_{i+1}$ is even

From equation (2) and (3) we have,

$$d'_i = r_{i+1} -1 -c_{i+1} \quad \text{and} \quad c_{i+1} = d_{i+1}$$

Using the above two conditions(i.e. $\sum_{i+2}^{n} d_i$ is even and $\sum_{i-1}^{n-1} c_{i+1}-c_{i+1}$ is odd) we will get,

$$d'_i = r_{i+1} -1 -d_{i+1} \quad \text{if } \sum_{i}^{n} r_i - r_{i+1} - d_i = \text{even}$$

Combining all the four cases we get,

$$d'_i = d_{i+1} \quad \text{iff} \quad d(i) \text{ is even}$$

$$= r_{i+1} - d_{i+1} -1 \quad \text{otherwise}$$

or

$$d'_i = d_{i+1} \quad \text{iff} \quad \sum_{i}^{n} r_i - r_{i+1} - d_i \text{ is odd}$$

$$= r_{i+1} - d_{i+1} -1 \quad \text{otherwise}$$

Hence proved.

Lemma 2 :

Suppose d is a murray integer ,c is the Gray coded integer ,A is the equivalent Gray coded integer obtained from c as defined above ,d' is the equivalent murray integer obtained from A ,then for all even i , we have

$$d'_i = d_{i-1} \quad \text{iff} \quad d(i) \text{ is even}$$

$$= r_{i-1} - d_{i-1} -1 \quad \text{otherwise}$$

or

$$d'_i = d_{i-1} \quad \text{iff} \quad \sum_{i+1}^{n} r_i + d_i \text{ is even}$$

$= r_{i-1} - d_{i-1} - 1$ otherwise

where n = no. of digits

Proof.

Since i is even, then from the above equations i.e., B and C, we have

$$A_i = c_{i-1} \text{ and}$$

$$R_i = r_{i-1} \qquad (4)$$

Further from equation D we have

$$d'_i = A_i \text{ iff } \sum_{i+1}^{n} A_i \text{ is even}$$

$$= R_i - 1 - A_i \text{ otherwise.}$$

Using condition (4) in the above equation we will get,

$$d'_i = c_{i-1} \text{ iff } \sum_{i+2}^{n+1} c_{i-1} \text{ is even} \qquad (5)$$

$$= r_{i-1} - c_{i-1} - 1 \text{ otherwise.}$$

Also by using equation (A) we can get ,

$$c_{i-1} = d_{i-1} \text{ iff } \sum_{i}^{n} d_i \text{ is even} \qquad (6)$$

$$= r_{i-1} - 1 - d_{i-1} \text{ otherwise}$$

Now we will use equation (5) and (6) to find a equation between d and d'. As usual we will consider four cases.

Case 1 : $\sum_{i}^{n} d_i$ is even and $\sum_{i+2}^{n+1} c_{i-1}$ is also even

From equation (5) and (6) we have,

$$d'_i = c_{i-1} \quad \text{and} \quad c_{i-1} = d_{i-1}$$

This implies $d'_i = d_{i-1}$.

The above two conditions (i.e. $\sum_{i}^{n} d_i$ is even and $\sum_{i+2}^{n+1} c_{i-1}$ is also even) can be joined to form a single condition . let us consider,

$$\sum_{i+2}^{n+1} c_{i-1} = \text{even}$$

or $$\sum_{i+2}^{n+1} d_{i-1} = \text{even}$$

$$d_{i+1} + d_{i+2} + \dots\dots + d_n = \text{even}$$

$$d_i + d_{i+1} + \dots\dots + d_n - d_i = \text{even}$$

But $\sum_{i}^{n} d_i$ is even, this implies, even $- d_i$ = even

i.e., d_i = even

Case 2 : $\sum_{i}^{n} d_i$ is even and $\sum_{i+2}^{n+1} c_{i-1}$ is odd

From equation (5) and (6) we have,

$$d'_i = c_{i-1} \quad \text{and} \quad c_{i-1} = r_{i-1} - 1 - d_{i-1}$$

Using the above two conditions(i.e $\sum_{i}^{n} d_i$ is even and $\sum_{i+2}^{n+1} c_{i-1}$ is odd) we will get,

$$d'_i = r_{i-1} - 1 - d_{i-1} \quad \text{if } d_i \text{ is odd}$$

Case 3 : $\sum_{i}^{n} d_i$ is odd and $\sum_{i+2}^{n+1} c_{i-1}$ is also odd

From equation (5) and (6) we have,

$$d'_i = r_{i-1} - 1 - c_{i-1} \quad \text{and} \quad c_{i-1} = r_{i-1} - 1 - d_{i-1}$$

This implies $d'_i = r_{i-1} - 1 - (r_{i-1} - 1 - d_{i-1})$

$$d'_i = d_{i-1}.$$

We can now find a single condition by joining the above two conditions (i.e. Σd_i is odd and Σc_{i-1} is also odd) . let us consider,

$$\sum_{i+2}^{n+1} c_{i-1} = \text{odd}$$

or $$\sum_{i+2}^{n+1} (r_{i-1} - 1 - d_{i-1}) = \text{odd}$$

$$\sum_{i+2}^{n+1} r_{i-1} - \Sigma d_{i-1} - \Sigma 1 = \text{odd}$$

$$\sum_{i+2}^{n+1} r_{i-1} + d_i - \Sigma d_i - (n-i) = \text{odd}$$

$$\sum_{i+2}^{n+1} r_{i-1} + d_i - \text{odd} - \text{even} + \text{even} = \text{odd}$$

$$\sum_{i+2}^{n} r_{i-1} + d_i = \text{even}$$

or $$\sum_{i+1}^{n} r_i + d_i = \text{even}$$

case 4 : $\sum_{i}^{n} d_i$ is odd and $\sum_{i+2}^{n+1} c_{i-1}$ is even

From equation (5) and (6) we have,

$$d'_i = r_{i-1} - 1 - c_{i-1} \quad \text{and} \quad c_{i-1} = d_{i-1}$$

Using the above two conditions(i.e $\sum_{i}^{n} d_i$ is odd and $\sum_{i+2}^{n+1} c_{i-1}$ is even) we will get,

$$d'_i = r_{i-1} - 1 - d_{i-1} \quad \text{if } \Sigma r_i + d_i = \text{odd}$$

Combining all the four cases we get,

$$d'_i = d_{i-1} \quad \text{iff} \quad d(i) \text{ is even}$$

$$= r_{i-1} - d_{i-1} - 1 \quad \text{otherwise}$$

or

$$d'_i = d_{i-1} \quad \text{iff } \sum_{i+1}^{n} r_i + d_i \text{ is even}$$

$$= r_{i-1} - d_{i-1} - 1 \quad \text{otherwise}$$

Hence proved.

Using the above two lemmas, the steps of the transformation from n to m (where "n" is the nth point on the scan1 and "m" is the mth point on the scan2), are

1. Convert n into the equivalent murray integer with the given radices,
2. Convert this integer into the equivalent murray integer of scan2 .
3. Convert back to the mth point on scan2.

4-2.2 Method 2 :

The efficiency for the method discussed above can be further improved for large complete scans. The first method discussed above is slow because there are so many operations which have to be applied such as gray code integer conversion, interchanging the digits etcetera. In this section we will present an algorithm which is usually faster than the one described above.

Let us consider a rectangle of size n*m, where n represents the number of columns and m represents the number of rows. The rectangle of size n*m can be scanned either by using scan1 or by using scan2, where scan1 and scan2 are described above.

Since murray polygons are space filling curves, they will pass through each and every point in a given space. Since each point will be visited only once we can mark all the points by giving them an integer number. Scan1 and scan2 will give a different numbering to the pixels (see Figure 4.2), since the direction is different.

The efficiency may now be improved if we deal straight with the numbers marked on each pixel (see Figure 4.2), to find the corresponding mth

The efficiency may now be improved if we deal straight with the numbers marked on each pixel (see Figure 4.2), to find the corresponding mth point in the next scan. Here we do not have to find the gray code conversion, digits interchange, etcetera.

Now we will define two lemmas, to get the above transformation. Before we discuss these two lemmas we will review some definitions already given.

Let $r_n, r_{n-1}, r_{n-2}, \ldots\ldots, r_1$ define the radices r_i associated with the digits d_i ($i = 1,2,\ldots.n$), such that for each i, $0 <= d_i < r_i$. As explained earlier the product of the first two radices will be equal to the number of pixels present in a tile i.e., (no.of pixels/tile is equal to $r_1 * r_2$), the product of the next two radices will be equal to the total no. of blocks.1(say), each of size $r_1 * r_2$ (i.e. size of a tile), the product of the next two radices will be equal to the total number of blocks.2(say), each of pixel size $r_1 * r_2 * r_3 * r_4$ and so on. For example:

If $r_1 = 3$, $r_2 = 3$, $r_3 = 5$, $r_4 = 7$, $r_5 = 3$ and $r_6 = 5$, then no. of pixels/tile is 9, total no.of blocks.1 of pixel size 9 are 35 and total no.of blocks.2 of pixel size 315 are15 .

From the following two lemmas , the first one finds the corresponding point in an image where a murray scan moves across the full width of the image before a unit change in the y-direction occurs. Here the whole image is considered as one large complete tile, whereas the second lemma finds the corresponding points in an image where a murray scan partitions the whole image into small tiles each of size $r_1 * r_2$ and the total number of such tiles in an image is equal to $r_3 * r_4 * \ldots\ldots * r_n$.

Lemma 3 :

Suppose a tile of size N*M is given, and that i is the ith point on the tile in scan1.Then the corresponding point on the tile in scan2 is given by, (scan1 and scan2 have their usual meaning),

j = M*A +B iff A is even

= M*A +M -1 -B otherwise

where, A and B are the row and the column numbers respectively, and,

N = r(1) and M = r(2)

Proof :

Consider an image of size N*M i.e., n columns and m rows. All the pixels in an image are numbered (see Figure 4.2). The top right numbers belongs to scan1 i.e., a horizontal scan and the bottom left numbers belongs to scan2 i.e., a vertical scan.

Let us consider scan1 first. From Figure 4.2, we can see that the start points for scan1 in each row are 0, N, 2N, ..., and (M-1)N. Each row has N points. If we divide each point in the first row by N then the divisor will be zero. For the second row the divisor will be 1 and and so on. If we multiply these divisors i.e., 0, 1, 2,........, and M-1 by N we will get the starting point for each row. Let us call the divisors 0, 1, 2,........,M-1 the y-values.

Similarly if we consider scan2, the start points for scan2 in each of the columns are 0, M, 2M, ...,(N-1)M. If we divide each point in the first column by M the divisor will be zero, similarly for second column, the divisor will be 1 and so on. The start point for each column can be

determined by multiplying m by these divisors i.e. 0, 1, 2, ...,N-1. Let us call them the x-values.

If the ith point on scan1 is given to us then it is very easy to get the start point for a row, by simply dividing the ith point by N to get the divisor and then multiplying it by N. To get the starting point for that column, we have to find the x-value first. Two cases can be considered,

1) y-value is even

Let us consider the 2Nth point on scan1. From Figure 4.2 it can be seen that the number 2N is obtained by multiplying N by the corresponding y-value of that row and then adding this to the corresponding x-value of the column. Here the y-value is 2 and the x-value is 0 which is equal to 2N rem N. Therefore for a given point (say ith) the x-value will be given as i rem N.

2) y-value is odd

Let us consider the (N+1)th point of scan1. The number N+1 is obtained by multiplying n by the corresponding y-value and then adding this to N-1-(x-value), since for all odd y-values the direction of the scan is reversed. Therefore for a given point (say ith) the x-value will be given as N-1- (i rem N).

So far we have discussed the case that if the ith point on scan1 is given then for an image of size N*M we have ,

y-value = i div no.of.columns

x-value = i rem no.of.columns, if y-value is even

no.of.columns -1 -(i rem no.of.columns), otherwise.

To get the corresponding jth point of scan2 we will consider two cases :

case 1 :

'x-value' is odd :

Suppose the ith point on scan1 is given(see Figure 4.2). Now we have to show that the corresponding point on scan2 is the jth . Since the x-value is odd this means that the curve i.e. scan2, is going from top to bottom. The starting point for the scan2 in that column will be equal to M*x-value. The number of places we have to move further is equal to M-1-(y-value) . So the corresponding point in scan2 is given by , M*(x-value) + M-1- (y-value) .

case 2 :

'x-value' is even :

Since the x-value is even, this means that the curve is going from bottom to top. The starting point for scan2 in that column will be M*x-value. The displacement will be given by the y-value. So the corresponding point in the scan2 is, M*(x-value) + y-value.

Combining all the results together it follows that if the ith point on scan1 is given then the jth point on scan2 will be given by,

$$j = M*A + B \text{ if A is even}$$

$$= M*A + m - 1 - B \quad \text{otherwise.}$$

Hence proved.

Before we discuss the next lemma we define some variables to be used.

Let $r_n, r_{n-1}, r_{n-2}, \ldots\ldots, r_1$ define the radices r_i associated with the digits d_i.$(i = 1,2,\ldots.n)$ and let i be the ith point in scan1.Then we define variables $a_1, a_2, a_3, \ldots\ldots, a_{n/2}$ as the tile numbers where,

$a_{n/2}$ is the tile number to which the ith point belongs, where the total number of tiles is $r_n * r_{n-1}$ with pixel size equal to $r_1 * r_2 * \ldots\ldots * r_{n-3} * r_{n-2}$.

$a_{n/2-1}$ is the tile number inside $a_{n/2}$ to which the ith point belongs, where the total number of tiles is $r_{n-3} * r_{n-2}$ with pixel size equal to $r_1 * r_2 * \ldots\ldots * r_{n-5} * r_{n-4}$.

a_1 is the tile number inside a_2 to which the ith point belongs, where the total number of tiles is $r_2 * r_1$ with pixel size equal to $1*1$.

Lemma 4 :

Let $r = r_n, r_{n-1}, r_{n-2}, \ldots\ldots, r_1$ be the radices r_i associated with the digits d_i.$(i = 1,2,\ldots.n(=2k$, where k is an integer)), such that for each i, $0 <= d_i < r_i$ and suppose an image of size l*b (where 'l' is the length of the block and b is the breath of the block) is given. If N, is the Nth point on the image in scan1, then the corresponding M^{th} point on the image in scan2 is given by $M_n = x_1 + r_1 * r_2 * x_2 + \ldots\ldots + r_1 * r_2 * \ldots\ldots * r_{n-4} * x_{n/2-1} + r_1 * r_2 * \ldots\ldots * r_{n-2} * x_{n/2}$

$$M_n = I_2 + I_4 + I_6 + \ldots\ldots + I_n \qquad (1)$$

where $I_n = r_1 * r_2 * \ldots\ldots * r_{n-2} * x_{n/2}$

r_0 = 1, and

$x_1, x_2, x_3, \ldots, x_{n/2}$ are the corresponding points on scan2 when the corresponding points i.e., $a_1, a_2, a_3, \ldots, a_{n/2}$ on scan1 are given.

Proof :

To prove this lemma for each positive even integer n, let us define P(n) to be the proposition $M_n = I_2 + I_4 + I_6 + \ldots + I_n$.

Suppose n = 2.Then equation(1) gives,

$$\text{L.H.S} = I_2$$

or $$\text{L.H.S} = x_1$$

$$\text{R.H.S} = M_2$$

which is clearly true.Since the total number of radices are two, the whole image is considered as *one* large complete tile and hence for a point a_1 on scan1 we will get x_1 the only point on scan2(refer *lemma 3).*

Suppose now that P(2n) is true for n <= k. In particular,

$$M_{2k} = I_2 + I_4 + I_6 + \ldots + I_{2k}$$

is true . That is P(2k) is true.

But, $$M_{2k+2} = M_{2k} + I_{2k+2}$$

or $$M_{2k+2} = I_2 + I_4 + I_6 + \ldots + I_{2k} + I_{2k+2}$$

which implies that the statement P(2k+2) is true. Therefore P(2k) implies P(2K+2). Hence since P(2) is true, the result follows by the principle of mathematical induction.

4-3 Implementation of Scan Conversion :

4-3.1 Data Structure :

The data structure for an algorithm plays an important role, since the efficiency of the algorithm depends upon that. The main data structure to be used in the following algorithms contains five main items,

runlength

Sum

Colour

left pointer

right pointer

we define our structure as,

structure int.list*(***int** *runlength,Sum;* **string** *Colour;* **pntr** *left,right)*

where *int.list* is the name for the structure and is of type *pntr*, *runlength* and *Sum* are of type *integer, Colour* is of type *string*, and finally the items *left* and *right* are of type *pointer.* The pointer variables are called *Link* . Each *structure* is linked to a succeeding *structure* by the member *right* .

The field *runlength* stores the successive runlengths which are obtained after scanning an image.The field *Sum* records the number of points used before a particular cell. The field *Colour* records whether the runlength corresponds to a BLACK area or to a WHITE area. If it corresponds to a white area then we put "w" otherwise "b". The field *right* points at the next cell and the field *left* points at the left cell. If no cell is present to the right or to the left side then the pointers will be set to nil. A pictorial representation for the structure with links is shown in Figure 4.3.

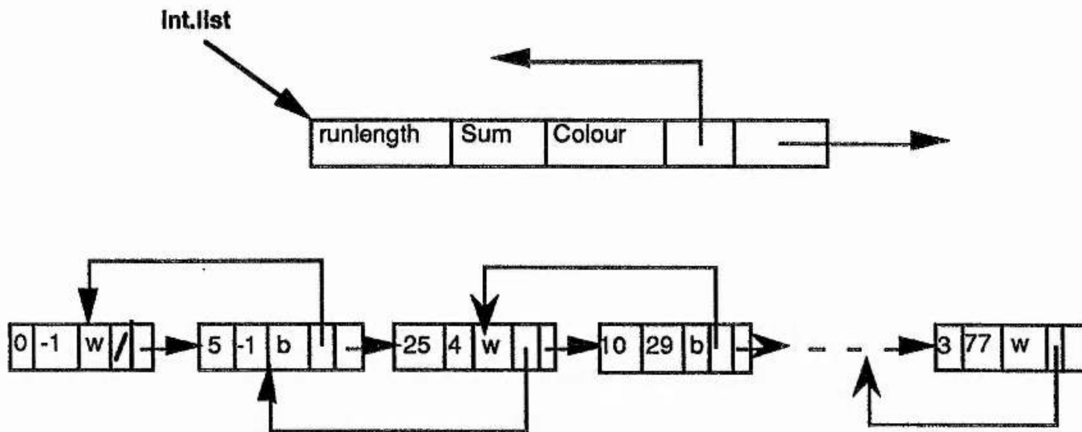


Figure 4.3. Pictorial representation for the structure with links. The third cell records that there are 25 consecutive WHITE pixels. The Sum, which is 5, records that 5 pixels are used before this cell.

Note: In the first cell the item Sum is assigned value -1, since our starting point in an image is zero (see Figure 4.2).

The field *Sum* plays an important role in implementing these algorithms. The main advantages of using this item can be summarized as :

1) It can be used to find the starting point for a particular cell. For example in Figure 4.3 consider the second cell. Here the *runlength* is 5 and the *Sum* is -1 and the *Color* of that cell is "b". Since Sum = -1 i.e., the number of points used before this cell is zero, hence the starting point for this cell will be zero. Further the color is black implies that the pixel number 0,1, 2, 3, and 4 are BLACK in color. Similarly in third cell the pixel numbered from 5 to 29 are WHITE in color.

2) It can be used to draw the image back on the screen quickly. Since the starting point of each cell is known to us, we can find the coordinate values for the start point and we can easily draw the image by considering only the BLACK cell.

3) The number of cells in a list can be reduced by removing all the white cells from the list. The new list will contain only the BLACK cells. If

we have to process the list, (for example if we have to scale the image we need both WHITE and BLACK cells), the WHITE cells can be obtained using the information stored in the BLACK cells.For example in Figure 4.4, consider the second cell. The *runlength* is equal to 10(say r1) and *Sum* is equal to 30(say s1). Suppose now we want to find the WHITE cell preceeding it. From the first cell the *runlength* is equal to 5(say r2) and *Sum* is equal to 0(say s2). We use the first two cells to find the WHITE cell in between. The *runlength* for this WHITE cell will be equal to s1 -s2 -r2 and the *Sum* will be equal to s2 +r2. Similarly for other WHITE cells.

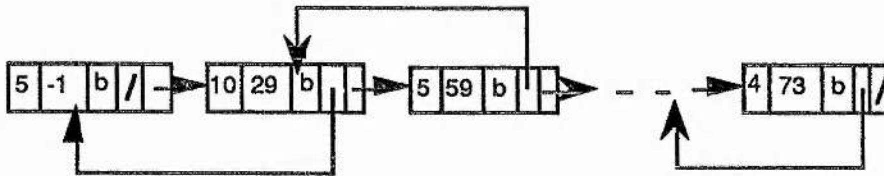
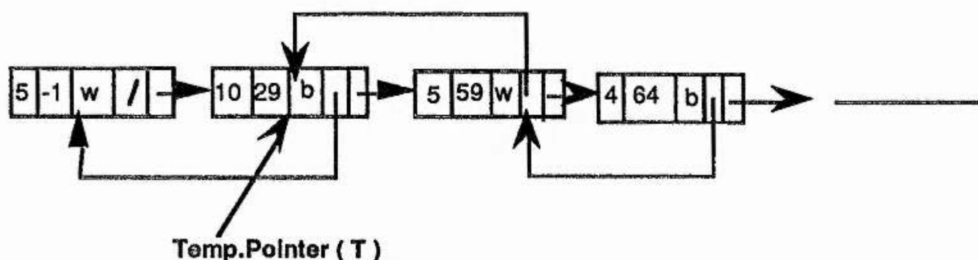


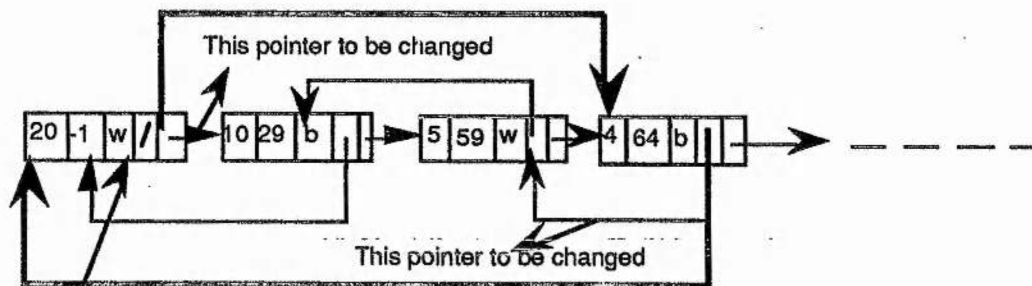
Figure 4.4. New list obtained after removing all the WHITE cells from the list given in Figure 4.3.

The third item in our data structure is the colour information. This colour information can be removed by assuming that the first cell will always corresponds to the white cell. But in some cases where we need to move inside the list, this information will be of great help. We will see this in the next chapters. Since same list is used for different operations, therefore all the list in the database have the colour information for each cell.

The left pointer helps in increasing the mobility inside the list. For example consider the list given below,



Suppose our temporary pointer is at the second cell as shown above. If the colour for that cell has changed to "w" then we need to add the two neighbouring WHITE cell to that to form a single cell. Using the left pointer we can go to the left cell. The three cells can then be merged together to form a single cell.The pointer can be changed thereafter, see below.



Temp.Pointer (T)

Initially the Temp.Pointer was at the 2nd cell(see above). By using the command T = T(left), the Temp.Pointer will now point at the 1st cell. The runlength for the first cell will now be equal to the sum of the runlengths of the three cells i.e., 20. Sum for this cell will be -1. The link next will now point at the 4th cell and the left pointer of the cell will now point at the 1st cell (shown dark).

4-3.2 Scanning :

Here an image is taken, which is stored in the database and then using murray polygons either horizontal or vertical, we scan the whole image. The collection of runlengths and the murray radices used to scan the image are then stored in the database.The procedure to store an image or a list in a database is given in chapter 3. Before the scanning is done we have to decide about the murray radices to be used.

To get the murray radices, we need to know the dimensions for the image. Generally the dimensions for an image are known to us but if not then the standard function *X.dim* and *Y.dim* (see PS-algol[Carrick, Cole, and Morrison(1987), and Morrison(1988)], can be used to get the x and the y dimensions of an image. Once we know the x and the y dimensions, murray radices can be defined. The only point to remember is that the product of the x-radices should be equal to the x dimension of the image. Similarly for the y-radices.

4-3.3 Algorithms :

Here two algorithms, one which uses method 1 and the other which uses method 2, are given.The results obtained by both the algorithms on different images are compared and are given in the next section. The two procedures corresponding to two lemmas (i.e., lemma 3 and lemma4), described above are given below,

```
! This procedure finds the corresponding points in a tile.
! The input is the ith point given on scan1 and the x and
! the y dimensions for the tile.
let pnt.in.scan2 = proc(int i,n,m -> int)
begin
   let y := i div n
   let x := if y rem 2 =  0  then i rem n
                             else n-1-(i rem n)
    let z := if x rem 2 = 0  then m*x+y
                             else m*x+m-1-y
    z  ! the corresponding point on the second scan is z.
end
```

```
! This procedure finds the corresponding points in a block having
! smallest tile of size x-dimension * y-dimensions.
! The input is the ith point given on scan1 and the murray radices.
! The vector Pix are the sizes of the corresponding blocks, e.g.
! Pix(1) = r1*r2, Pix(2) = r1*r2* r3*r4  and so on.
! The vector block gives the position of the point in an image.
! The vector mult.factor  gives the point x1,x2, .... (see sec 4-2.2, lemma2)
let scan.conversion = proc(*int r,Pix,block,mult.factor;int i -> pntr)
begin
   let j := 1;let C := i
   for i = upb(Pix) to 1 by -1 do      ! This statement will give the
            {block(j) :=i div Pix(i) ! position of the point relative to the
             i := i-Pix(i)*block(j)  ! blocks of sizes Pix(i) .
             j := j+1}
   j:=1
   for i = upb(r)-1 to 3 by -2 do      ! It gives the point x1,x2, ..... .
            {mult.factor(j):=pnt.in.scan2(block(j),r(i),r(i+1))
             j:=j+1}
   let tile.start.pnt := 0; let x:=0
   j:=1
   for i = upb(Pix) to 1 by -1 do      ! This gives value A.
            {tile.start.pnt:=tile.start.pnt+block(j)*Pix(i)
             x:=x+Pix(i)*mult.factor(j)
             j:=j+1}
   let y := C-tile.start.pnt
   let z := pnt.in.scan2(y,r(1),r(2))
   let M := x+z
end
```

Theory :

The algorithm takes a list of runlengths which has been stored in the database.It will deal only with the runlengths and secondly it deals only with the black cells. Time and space is saved by not considering the white cells.

Let $r_1, r_2, \ldots, r_n$ be the murray radices. The number of points in a image will then be given as the product of all murray radices, i.e.,

$$\text{no.of.pixels} = r_1 * r_2 * \ldots * r_n$$

We are interested in finding the runlengths corresponding to another scan. Initially we will assume that the linked list (say list.2) for scan2 has only one cell with the *runlength* equal to the number of pixels in an image, *Sum* equal to zero and the *color* for all the pixels is white i.e.,"w". From the linked list which is obtained after scanning the image in the horizontal direction i.e. scan1, we look for those cells which are black in color. Since we have the record for the number of pixels used before that cell i.e., *Sum*, we can easily find the start point for that cell and the number of pixels of that color i.e., *runlength*. Using either of the two methods discussed above, depending upon the scan used for scanning the image, the corresponding point on the second scan can be determined. *Lemma 3* is used when our whole image is represented by a single block i.e., we scan the image using a linear horizontal murray scan. When the image is represented by a collection of small tiles, we use *Lemma 4.* For each black point we will find the corresponding point in the second scan and the list.2 will be adjusted thereafter.

3-3.4. Comparison between the two algorithms:

Table 4.2. Comparison betweeen the two methods

An Image	Radices $r_1\ r_2\ r_3\ r_4\ r_5\ r_6$	Number of Black Pixels	Number of White Pixels	Method 1 (time)	Method 2 (time)
Image a	11 11 9 9	2875	6926	1.10 secs	0.59 secs
Image a	11 11 3 3 3 3	2875	6926	1.12 secs	1.00 secs
Image b	11 11 9 9	3134	6667	1.08 secs	0.55 secs
Image b	3 3 3 3 11 11	3134	6667	1.30 secs	1.20 secs
Image e	7 9 5 7	608	1597	0.22 secs	0.17 secs
Image e	7 7 3 5 3 1	608	1597	0.27 secs	0.25 secs
Image c	11 11 3 3 3 3	6246	3555	2.10 secs	2.00 secs
Image f	39 39 3 3	1810	11879	3.10 secs	3.09 secs
Image f	13 13 3 3 3 3	1810	11879	3.15 secs	3.10 secs

The different images which are given in chapter 3 are considered to compare the two algorithms discussed above. The result obtained is shown in Table 4.2. Method 2 is found to be faster than that of the method 1. The reason is, in the first method the conversion from murray digits to gray code integer, gray code conversion etcetera are more time consuming. In the case of the second method the algorithm deals only with the numbers marked on each pixels (refer section4-2.2).

3-3.5 Comparison between Linear murray scan and General murray scan:

An Image	Radices $r_1 r_2 r_3 r_4 r_5 r_6$	Number of Black Pixels	Number of White Pixels	Method 1 (time)	Method 2 (time)
Image a	99 99	2875	6926	1.08 secs	0.55 secs
Image a	11 11 3 3 3 3	2875	6926	1.12 secs	1.00 secs
Image b	99 99	3134	6667	1.20 secs	1.05 secs
Image b	3 3 3 3 11 11	3134	6667	1.30 secs	1.20 secs
Image c	99 99	6246	3555	2.29secs	2.17secs
Image c	11 11 3 3 3 3	6246	3555	2.10 secs	2.00 secs

Table 4.3. Comparison betweeen the Linear murray scan and the General murray scan.

As shown in Table 4.3, the linear murray scan takes less time than the one which breaks an image into the small tiles. In all the cases except the last one (i.e image c),the result is in favour of linear murray scan.The reason for this is, in the case of linear murray scan the whole image is assumed to be of a single block whereas in the general murray scan an image is divided into small tiles each of size $r_1 * r_2$ and hence finding a particular point of scan1 in scan2 will take longer. In the case of a linear murray scan we will use only lemma 3 (section 4-2.2), whereas in the case of the general murray scan we have to use both the lemmas given in section 4-2.2. In the case of image c (see Table 4.3), a linear scan takes more time than that of standard murray scan. The reason is that the number of black runlengths obtained by the two scans i.e., a general murray scan and a linear murray scan. With a linear scan the total number of black runlengths obtained is 255, whereas with general

murray scan this number is 154. Since a general murray scan is frequently changing direction, there will be some long runlengths belonging to an image, whereas in the case of a linear scan these long runlengths may be broken into a large number of runlengths. Hence this time of processing one large black runlength corresponding to an area will be less than the runlengths which are originated from the same area.

From the above discussion we can conclude that the efficiency corresponding to a linear scan and the general murray scan depend upon the image. But as discussed above, a linear scan has only one tile (i.e. image) to process whereas a standard murray scan has $r_3*r_4*.......*r_n$ tiles to process, so in most of the case a linear scan will be faster than the standard murray scan. But a large number of radices will have a better chances of exploiting the coherence between the pixels and hence may result in better compression.

4-4 Scaling :

4-4.1 Introduction :

Scaling is the process of expanding or reducing the dimensions of an image. The factor by which an image is enlarged or reduced is called the scaling factor and the operation that changes the size is called scaling. Positive scaling constants s_x and s_y are used to describe changes in length with respect to the x-direction and to the y-direction.If the scaling factor is greater than one then this indicates an expansion of length, and if it is less than one, then a reduction of length. In the case of a picture the scaling effects can be obtained by multiplying the x and the y co-ordinate of every point in the picture by their corresponding scaling factors. Figure 4.6 shows scaling transformation with scaling factors $s_x = 2$ and $s_y = 1/2$.

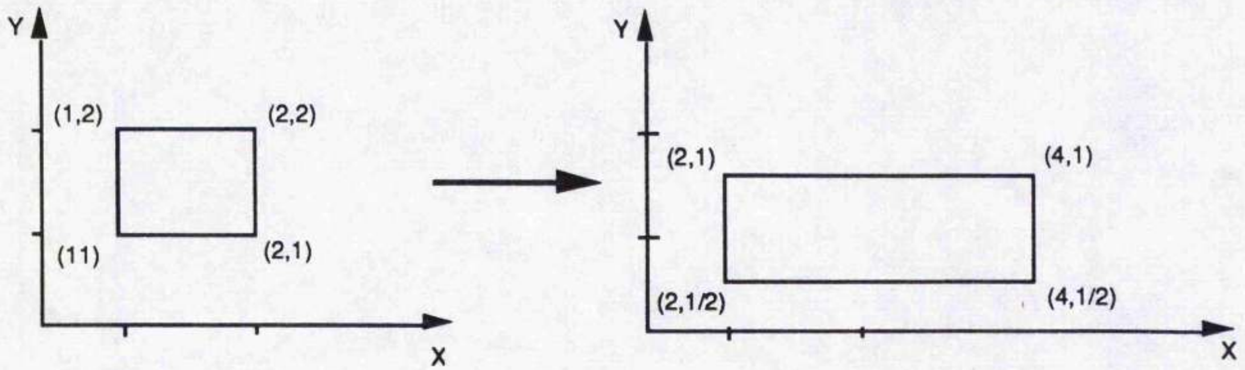
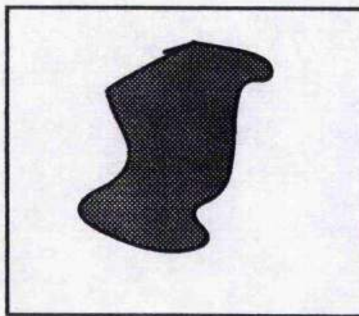
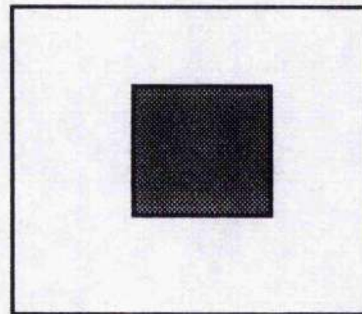


Figure 4.6. Scaling transformation with scaling factor Sx = 2 and Sy = 1/2

In the case of a picture the new co-ordinate can either be an integer or a real number. But if we want to scale an image then our scaling factor should be such that it gives integer co-ordinates when multiplied by the co-ordinates of the image. Exceptionally, a real factor may be used provided it gives an integer co-ordinate. For example $s_x = 1/2$, $s_y = 1/2$ and $(x,y) = (2,4)$. In this case the new co-ordinate will be (1,2), which is acceptable. A scaling factor to be real depends very much on the images also. Consider two images given below,



(a)



(b)

In the case of (a) the aliasing effect will be there, since in some case we have to truncate a real number to get an integer number. But since the

shape of the image is very irregular, this effect will not be easily seen. Whereas in case of (b) this effect will be very clear since the image is very smooth.Since each point on an image is going to be scaled, the time of completion will be very large. The computer time can be reduced if we work only on the runlengths encoding of an image, since a large number of pixels belonging to a runlength will be dealt with together. Further if a runlength is not completely divisible by the scaling factor then the remainder term can be added to the next runlength, thus keeping the shape of the image. Here murray scan techniques are used to scale the images. Different images are scaled up and down and the result corresponding to the different algorithms which uses murray techniques are presented. Finally the algorithm has been compared with those obtained from linear runlength encoding and quadtree encoding.

4-4.2 Scaling using murray polygons and its implementation :

In the next following paragraphs we will discuss an algorithm which uses murray techniques, to scale the images up and down. Later on some modifications to the algorithm are discussed in detail.

Let r_1/r'_1 be the rational scaling factor which is to be applied in the x-direction, where r'_1 is the x-dimension of initial smallest tile and r_1 is the x-dimension of final smallest tile. The runlengths are obtained by using a murray scan, whose initial movement was in the x-direction. For scaling, each runlength is multiplied by the factor r_1/r'_1. The remainder term will be added into the next runlength. That is if r_i is the runlength and R_{i-1} was the previous remainder then the new runlength is $(r_i*r_1+R_{i-1})$ div r'_1 and the remainder term which is to be added to the next runlength is given by $(r_i*r_1+R_{i-1})$ rem r'_1. For example,

let 4,3,5, be the runlength encoding for an image where $r'_1 = 3$ and $r_1 = 5$. The scaling factor is given by r'_1/r_1. Let x_1, x_2, x_3 are the new runlengths for the enlarged image, where,

$$
\begin{aligned}
\text{Remainder} &= 0 \\
x_1 &= (4*5 + \text{Remainder}) \text{ div } 3 \\
&= 6 \\
\text{Remainder} &= (4*5 + \text{Remainder}) \text{ rem } 3 \\
&= 2 \\
x_2 &= (3*5 + 2) \text{ div } 3 \\
&= 5 \\
\text{Remainder} &= (3*5 + 2) \text{ rem } 3 \\
&= 2 \\
x_3 &= (5*5 + 2) \text{ div } 3 \\
&= 9 \\
\text{Remainder} &= 0
\end{aligned}
$$

The changed runlengths are 6,5,9.

Note: *When we reach the corner of a tile the remainder term will be zero. Consider an expression ,*

$$x = a/b$$

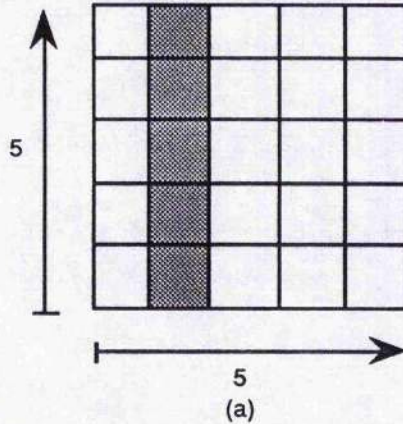
Here the remainder term will be zero if the quantity 'a' is either zero or it is multiple of b. Let us assume that the initial length of an image is I and the final length is F , where F > I. Since F > I this implies we are distributing F-I pixels equally in I pixels thus keeping the final size to F. Further if there is a remainder term we add this remainder term to the next runlength, without loosing any information. Hence when we reach the corner of a tile the remainder term will be zero . A lemma has discussed later to prove this .

The procedure is given below, which takes a sequence of runlengths and then scales them according to the scaling factor.

```
! This procedure takes two lists and returns
! a linked list with name list.
let enter = proc(pntr list,new -> pntr)
if list = nil then new else
begin
   let temp := list
   while temp(next) ~= nil do
   temp := temp(next)
   temp(next) := new
   list
end

! The input is a file containing all the runlengths,
! The scaling factor is equal to R/r.
! The output is a list with the scaled runlengths.
structure int.list(int run; pntr next)
let scaling.1 = proc(pntr list;file f;int R,r -> pntr)
begin
   let remainder := 0
   while ~eoi(f) do
   begin
      let x := readi(f)    ! This takes a integer from the file named 'f'.
      let x.scale := (x*R+remainder) div r
      list := enter(list,int.list(x.scale,nil))
      remainder := (x*R+remainder) rem r
   end
   list
end
```

With some images this approach is not very appropriate. For example if we consider a VLSI design as an image to scale down, then we see that the vertical lines are not straight after scaling. Let us consider a small example; an image of size 5*5 and the runlengths corresponding to that image is given in Figure 4.7. Let the initial x-radices be 5 and the final x-radices be 7. Therefore the scaling factor will be given by 7/5, which is greater than one i.e., an expansion.



the image is scanned using a horizotal murray scan. the corresponding runlengths are given below,

w b w b w b w b w b w
1 1 6 1 2 1 6 1 2 1 3

using the above approximation and with the scaling factor equal to 7/5 the new sequence of runlengths will be given as,

1*7 div 5 = 1 (2)

(1*7 +2) div 5 = 1 (4)

and similarly for the other runlengths . the new sequence now is ,

w b w b w b w b w b w
1 1 9 1 3 1 9 1 3 1 5

the new dimensions for the image are now 7*5. if we draw the image back on the screen using the new sequence of runlengths obtained after enlarging the image in the x-direction we will find that the vertical line is not striaght.

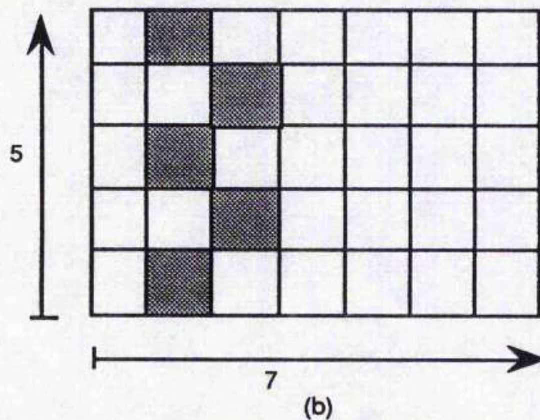


Figure 4.7 . The image (a) has been scaled using the approximation given above, the effect can be easily seen in (b).

The reason for that is the remainder term which we are adding to the next runlength. If we examine the scaling factor which is 7/5, states that we are replacing 5 pixels by 7 pixels. We see that each pixel out of 5 pixel is increased by the value equal to 7/5. If we assume that each pixel is made up of 5(say) parts then our remainder term will corresponds to these parts, for example, if the remainder term is 4(say), this means 4 parts out of 5 parts belongs to the previous runlength, which is very near to 5 i.e., one complete pixel.

The previous method may be now further improved by recording the nearest integer in the new runlength and passing a positive or negative carry to the next part of the calculation. The procedure is given below .

```
! The input is a file containing all the runlengths,
! The scaling factor is equal to R/r.
! The output is a list with the scaled runlengths.
structure  int.list(int run; pntr next)
let scaling.1 = proc(pntr list;file f;int R,r -> pntr)
begin
        let remainder := 0
        while ~eoi(f) do
        begin
           let x := readi(f)    ! Input is from the file named 'f'.
           let x.scale := (x*R+remainder) div r
           remainder := (x*R+remainder) rem r
           if remainder > r div 2 do
            { x.scale := x.scale+1; remainder := remainder-r }
           list  :=  enter(list,int.list(x.scale,nil))
        end
        close(f)
        list
end
```

Consider again the image given in Figure 4.7(a), the results are shown below,

Using the above approximation and the scaling factor equal to 7/5, the new sequence of runlengths will be given as,

$1*7 \text{ div } 5 = 1_{(2)}$

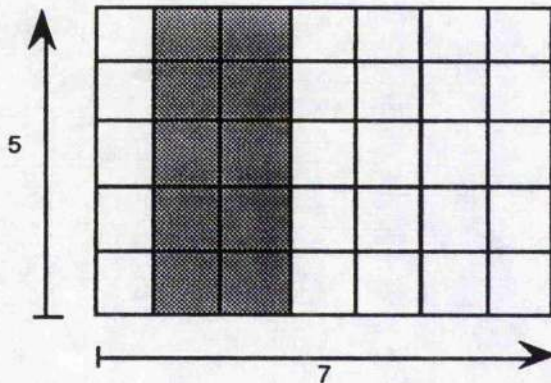
$(1*7 + 2) \text{ div } 5 = 1_{(4)} = 2_{(-1)}$ Since the remainder term is 4, which is nearer to 5 than to 0, hence we will add one to the new runlength and a negative carry to the next runlength.

Similarly for the other runlengths, the new sequence now is,

w b w b w b w b w b w

1 2 8 2 2 2 8 2 2 2 4

The new dimensions for the image is now 7*5. If we draw the image back on the screen using the new sequence of runlengths, obtained after enlarging the image in the x-direction, we will find that the vertical line is straight (see below),



To prove the above result a theorem is given. Before we define the theorem we will define few lemmas.

Lemma 5 :

Suppose r_{n-1} , r_n be two adjacent runlengths. Then the round off error which is passed on after scaling the two run lengths consecutively is the same as if the two runlengths are added together i.e. $(r_{n-1} + r_n)$, and the same scaling performed.

Proof :

Suppose,

n = x dimension of initial smallest tile,

P = x dimension of final smallest tile,

i.e. P/n = x-scaling factor.

$R(n-2)$ = round off error passed on from the calculation on r_{n-2} and is less than or equal to (n div 2).

The round off error which is passed on after the second runlength i.e., r_n is given as $(P^*r_n + (P^*r_{n-1} + R(n-2)) \text{ rem } n) \text{ rem } n$. (1)

The round off error which is passed on when the two runlengths are joined together is given as $(P^*(r_n + r_{n-1}) + R(n-2)) \text{ rem } n$. (2)

We have to show now remainder term (1) is equal to remainder term (2). Which is true since , A rem B = (A rem B) rem B, where A, and B are integers. Hence the remainder term (2) can be written as,

$$(P^*r_n + P^*r_{n-1} + R(n-2)) \text{ rem } n.$$

or $$(P^*r_n + (P^*r_{n-1} + R(n-2))) \text{ rem } n.$$

or $$(P^*r_n + (P^*r_{n-1} + R(n-2)) \text{ rem } n) \text{ rem } n.$$

Which is equal to remainder term (1). Hence proved.

Lemma 6 :

Starting at a vertex of a horizontal line, then the round off error from the last point of that line will be zero.

Proof :

Suppose,

n = x dimension of initial smallest tile,

P = x dimension of final smallest tile,

i.e. P/n = x-scaling factor.

$r_n, r_{n-1}, \ldots\ldots\ldots\ldots, r_1$ = the runlengths corresponding to a horizontal line of size n.

Using lemma 5, this follows easily by considering $r_n, r_{n-1}, \ldots\ldots\ldots\ldots, r_1$ runlengths as a single runlength of size n or, as n separate runlengths.

Theorem :

Suppose,

n = x dimension of initial smallest tile,

P = x dimension of final smallest tile,

i.e. P/n = x-scaling factor.

Rm(n-2) = round of error passed on from the calculation on r_{n-2} and is less than or equal in magnitude to (n div 2).

let r_i (i = 1 to N) and s_j (j = 1 to M) be the runlengths corresponding to two adjacent horizontal lines, where N and M are the integers.Then if, r_i and s_j are the two runlengths, corresponding to two horizontal(or vertical) lines, such that the start point of the first runlength, and the end point of the second runlength, have the same x-coordinate(or y-coordinate) say x, and adjacent y-coordinate(or x-coordinate) in the initial scan, then they will have the same x-coordinate(or y-coordinate) say x', and similar adjacent y-coordinate (x-coordinate) in the final scaled scan.

Proof :

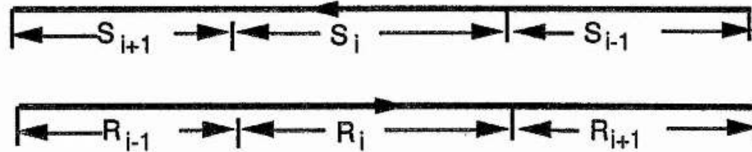
By Lemma 6, the accumulated round off error at the begining of an horizontal line segment is zero. Hence corresponding horizontal line segments will maintain their relative positions in the two scans. We have only therefore to prove the result for the x -coordinates.

Suppose the new scaled runlength corresponding to r_i is R_i and let S_j be the scaled runlength corresponding to s_j. To prove that the runlength Ri and S_j have the same x-coordinate and similar adjacent y-coordinate, we will compare the round off error received by the runlength Sj and the round of error received by the runlength R_{i+1}. The new runlengths will have the same x-coordinate and adjacent y-coordinate if and only if the round of error received is the same in magnitude for both the runlengths. Using Lemma 5, and Lemma 6, we now have to prove that,

$$S_{j-1} * P \text{ rem } n = (R_i * P + Rm(i-1)) \text{ rem } n,$$

or $$R_{i+1} * P \text{ rem } n = (R_i * P + Rm(i-1)) \text{ rem } n, \text{ (see below).}$$

or $$R_{n+1} * P \text{ rem } n = (R_n * P + Rm(n-1)) \text{ rem } n \qquad (A)$$



To prove this, for each positive integer n, let us define p(n) to be the proposition, $R_{n+1} * P \text{ rem } n = (R_n * P + Rm(n-1)) \text{ rem } n$

Suppose n = 1. Then equation(A) gives,

L.H.S = 0

R.H.S = 0

which is clearly true. Suppose now that p(n) is true for n <= k. In particular ,

$$R_{k+1} * P \text{ rem } k = (R_k * P + Rm(k-1)) \text{ rem } k$$

is true. That is p(n) is true. But,

$$\text{R.H.S} = (R_{k+1} * P + Rm(k)) \text{ rem } (k+1)$$

$$= (R_{k+1} * P + ((R_1+R_2+........+R_k)*P + 0)) \text{ rem } (k+1)$$

$$= (R_1+R_2+........+R_{k+1})*P \text{ rem } (k+1)$$

$$= (-R_{k+2}*P + ((R_1+R_2+........+R_{k+1})*P \text{ rem } (k+1)) \text{ rem } (k+1)$$

(A rem B = (A rem B) rem B)

$$= (-R_{k+2}*P + 0) \text{ rem } (k+1)$$ (using lemma 1)

$$= |\text{L.H.S}|$$

Which implies that the statement p(k+1) is true. Therefore p(k) implies p(k+1).

Hence since p(0) is true, the result follows by the principle of mathematical induction.
Hence proved.

To enlarge the images the scaling factor can take any value greater than 1. If we have to contract an image our scaling factor should be less than one. The images can be contracted only upto a certain extent. If our scaling factor is less than one,then two cases are possible,

1) It will contract an image without distorting the image.

2) It will deform the shape of the image either,

   i) by turning some of the white runlengths to zero. Here we will assume that the image is overscaled, and there is no way to bring back the original shape from the new collection of runlengths. If we are dealing with images which are not very smooth, for example an image of a tree, etcetera, then this method of contracting an image may be advantageous in approximately compressing an image and thus reducing the storage space, (refer Buntin(1988)).

   ii) by turning some of the black runlengths to zero. Here we may obtain the shape of the image which is very similar to the original one, by giving a runlength of one to the black cell from the white cell which is a left or right neighbour to it.This is explained later.

In case 2(ii) the black runlength which has turned to zero can accept a runlength of one either from the left cell or the right cell. The question is which cell to consider to give one value to the black cell. Many assumptions can be made, we can assume in the beginning that the black cell will always

accept a pixel from the right cell or from the left cell or if a white cell has two surrounding black cells which are turned to zero, then we can give one value to each black cell etcetera. The above assumptions are not always true, since we do not know which assumption to use and when.

Now we will discuss another approach which is more satisfactory than the previous assumptions. The theory behind this approach is same accept here we will discuss the cases as when to use right runlength and when to use the left runlength to increase the runlength of the black cell which has zero value. This idea has obtained by considering the path of the murray scan.

Consider an image of size n*m. Let $r_1r_2r_3r_4r_5r_6$ are the radices corresponding to the murray digits $d_1d_2d_3d_4d_5d_6$. Let $r_1 = 5$ and $r_2 = 5$, this means that the size for the smallest tile is 5*5. Since r_2 is equal to 5 means each tile will have five rows. We can assign a boolean value to each row . All the odd rows will have value T and all even rows will have F as the boolean value i.e., the first row will have value T (i.e. true), second row will have value F(i.e. false) and so on. Now we can say that, if a black runlength which has turned to zero is in a row which has value T then take a runlength of size 1 from the white cell which is right to that black cell and if the value is F then take a value 1 from the left white cell. To explain this, we will consider an example. Initially the correct scaled runlengths are obtained manually and then the above idea is used to compare the runlengths obtained. Consider the same image of size 5*5 in Figure 4.7(a). If we scale the image with the scaling factor to be less than 1 (say 3/5) then we will find that all the black runlengths are turned to zero, see Figure 4.8,

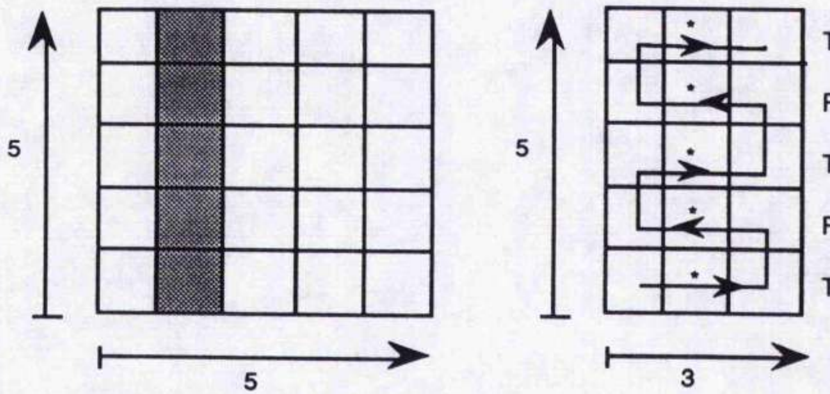


Figure 4.8. The image (a) has scaled down (b), turning all the black pixels to zero. The murray path has shown in (b) with the corresponding boolean value. "*" indicates that this square should be black to restore the image.

The new sequence of runlengths will be given as,

serial.number ---> 1 2 3 4 5 6 7 8 9 10 11

colour ---> w b w b w b w b w b w

runlength ---> 1 0 4 0 2 0 4 0 2 0 2

The 3rd runlength which is 4 is in between the two black cells which are turned to zero. If we want them to lie in a same column then it will be satisfactory to give one pixel each to the black cells from the 3rd runlength, thus decreasing it by two. The new runlengths will now be ,

1, 1, 2, 1, 0, 2, 0, 4, 0,2, 0, 2

If we use the idea of using boolean values for each rows as discussed above then the first black cell i.e., numbered 2 lies in a row which has value 'T' (see above) and according to the above statement we will take a runlength of size 1 from the one which is right to it, i.e 4. The new runlength will now be,

1, 1, 3, 0, 0, 2, 0, 4, 0,2, 0, 2

The second black cell numbered 4 which has value 0 lies in the row which has value 'F'(see above) and hence we will accept one from the left white cell numbered 3. The new runlengths are,

1, 1, 2, 1, 0, 2, 0, 4, 0,2, 0, 2

which is same as above. Same assumption can be assumed for the other black cells also. Sometimes an error can occur when dealing with a straight horizontal line. Due to the scanning pattern a murray scan can divide a line into two parts i.e. half part of the line lies in one tile and other half lies in the other tile see Figure 4.9(a). It has been noticed especially, with VLSI images, that when we scale down an image the two parts of a horizontal line which are not in the same tile are not in the straight horizontal line after substituting value one for the zero black cells. This problem can be solved by considering the same assumption as used above for the Figure 4.8. Only one point to remember is when we change the tiles the boolean value for the rows will also change i.e., if in the previous tile we are assuming that all the odd rows will have boolean value 'T' then in this tile they will have value 'F', same for even numbered rows. This is because the direction of a scan changes from tile to tile. For example, consider Figure 4.9(b) obtained after scaling the image (a), with the scaling factor less than 1(say 3/7).

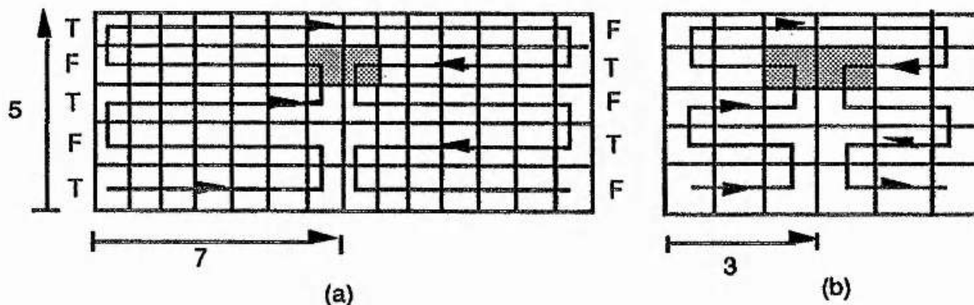


Figure 4.9 . Image (a) has scaled down with scaling factor equal to 3/7. The final image is shown in (b).

The runlength obtained for an image (a) are,

21, 1, 26, 1, 21

The corresponding runlength for the image (b) are ,

9, 0, 12, 0, 9

If you see the image (a) the first black belongs to the row having value 'F' and the second black cell belongs to the row having value 'T', i.e., both the cells will take one runlength from the 3rd cell which is12. The effect of changing the row values with the change in tiles can easily be seen above. The procedure put value of one to a black cell if it is zero is given below,

```
! The input is a list containing the black cells which are turned to zero.
! and a vector of boolean values i.e.  rows ( t ) . In case of scaling in
! vertical direction we will use a column vector. The integers r and R
! are equal to r1, and r2 i.e. first  two murray radices. The output is a
! new list with supplied one for zero cell.

let put.1.for.0.black = proc(pntr List ;*bool t ;int r,R -> pntr)
begin
   let temp := List        ! This is to keep the head name same as List.
   let sum := -1           ! This is used to find the row .
   let product := R*r
   while temp ~= nil do
   begin
      if temp ~= nil and temp(run) = 0 and temp(col) = "w" do
      begin          ! to check the value for the white cells.
         write"*******You have overscaled the image******'n"
         write"*******A white cell has turned to zero******'n"
         abort
      end
      if temp(run) = 0 do
      begin
         let i := (sum div r) +1
         temp(run) := 1
         ! Next statement gives a value of  one to the black cell which
         ! is zero according to the row value.
         if ~t(i)   then      {
```

```
                              temp(next,run) := temp(next,run)-1
                         temp(next,sum1) := temp(next,sum1)+1
                              }
                    else      {
                              temp(left,run) := temp(left,run)-1
                              temp(sum1) := temp(sum1)-1
                              }
      end
      sum := sum+temp(run)
      ! This statement is to change the value of the rows if
      ! the tile has changed.
      if sum >= product-1 do
      begin
         for i = 1 to R do
         t(i) := ~t(i)
         sum := sum rem product
      end
      temp := temp(next)
   end
   List
end
```

To scale the image in the y-direction the runs resulting from the first part need to be transformed into an essentially y-directional scan and a similar process applied. Using the *scan conversion* algorithms discussed above we can convert the runs from a horizontal murray scan to a vertical murray scan and the new runs can be scaled in the y-directions also.

Figure 4.10 shows a scaling up and down of the images. In the case of VLSI images (or any image), we will keep on scaling down an image until there is a dead short i.e., when few white pixels will turn to zero. At this stage the programme will stop giving the message "*You have overscaled the image* ". The result of scaling up and down of different images is given in Table 4.3.

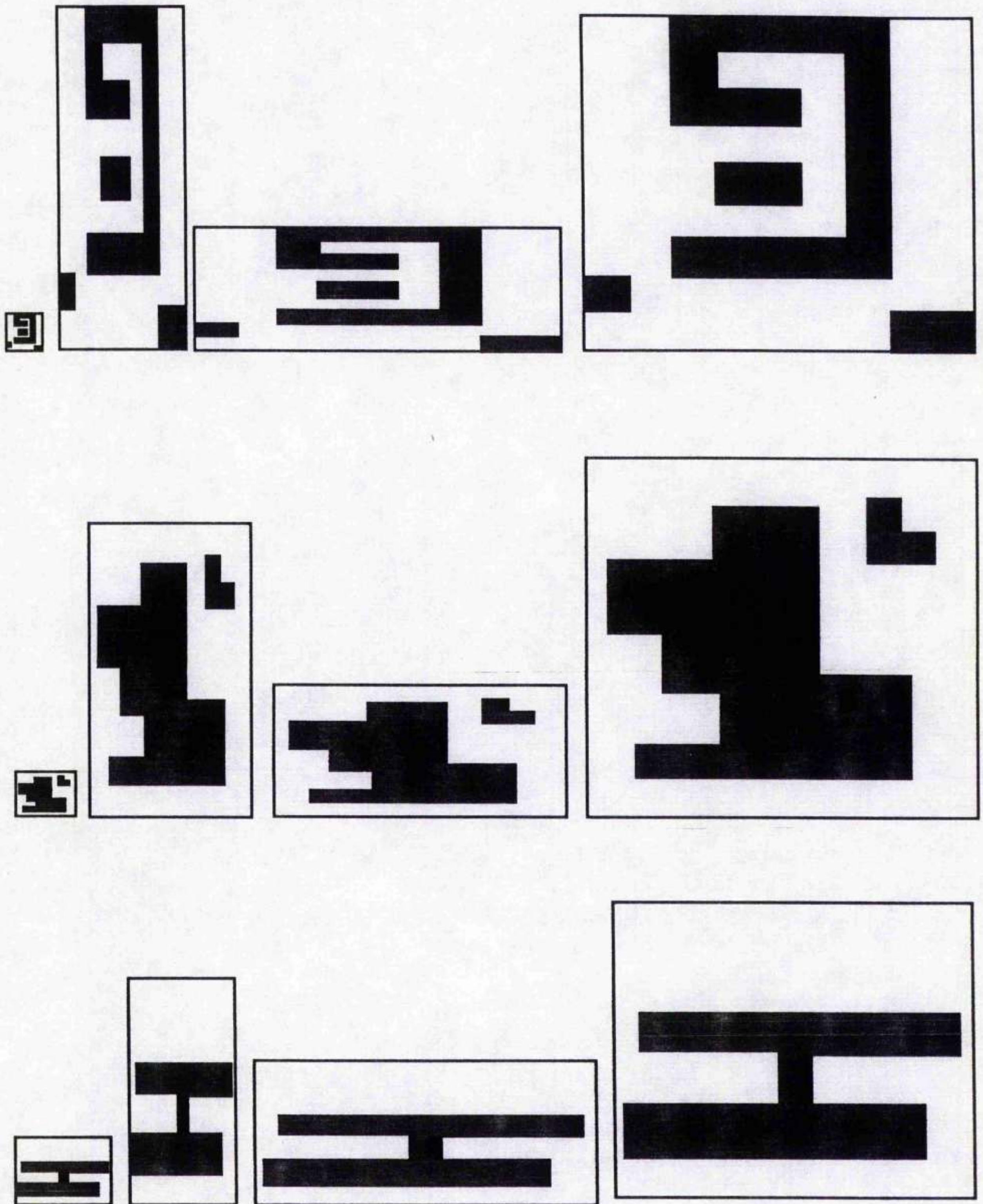


Figure 4.10. Scaling effect on the different images.

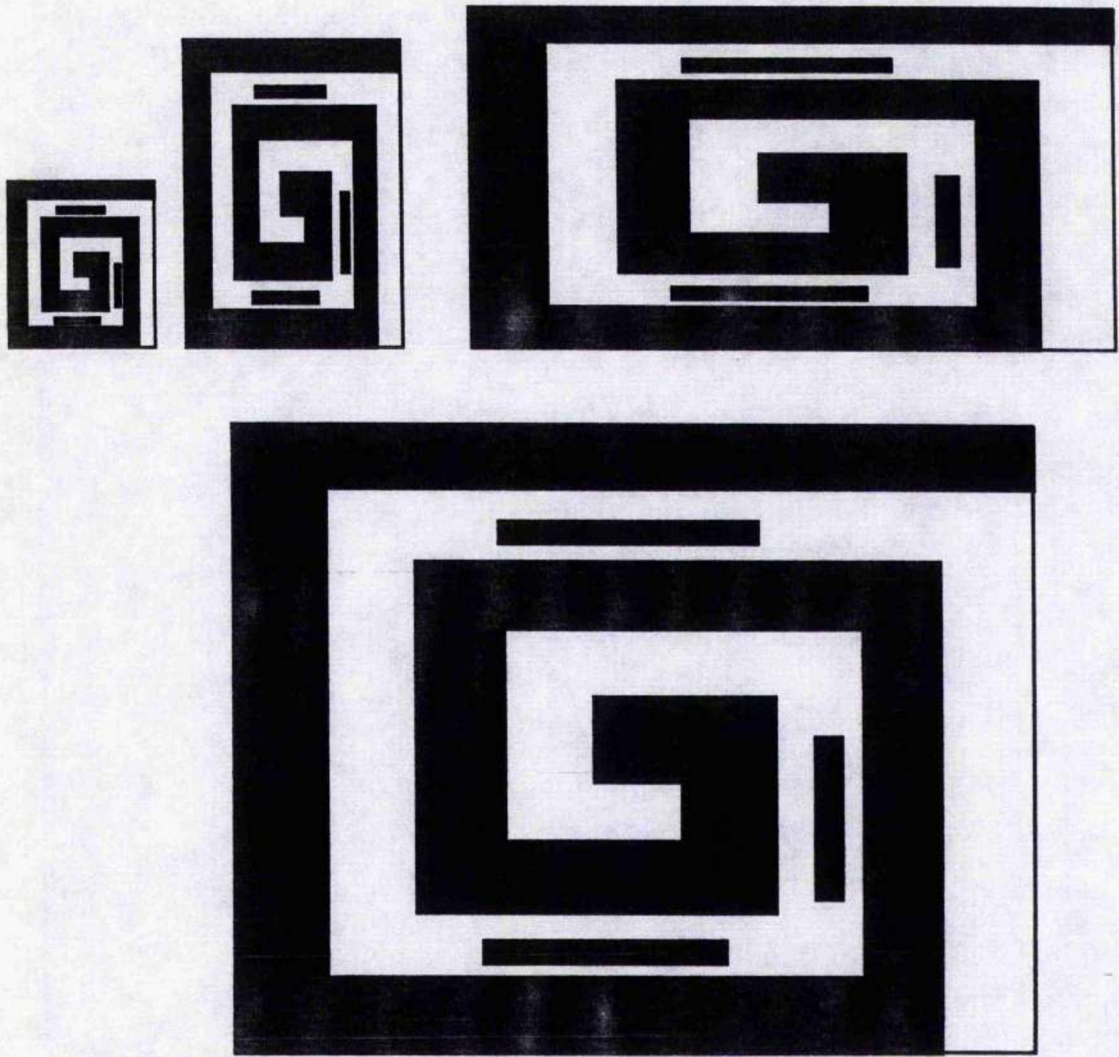


Figure 4.10[contd].

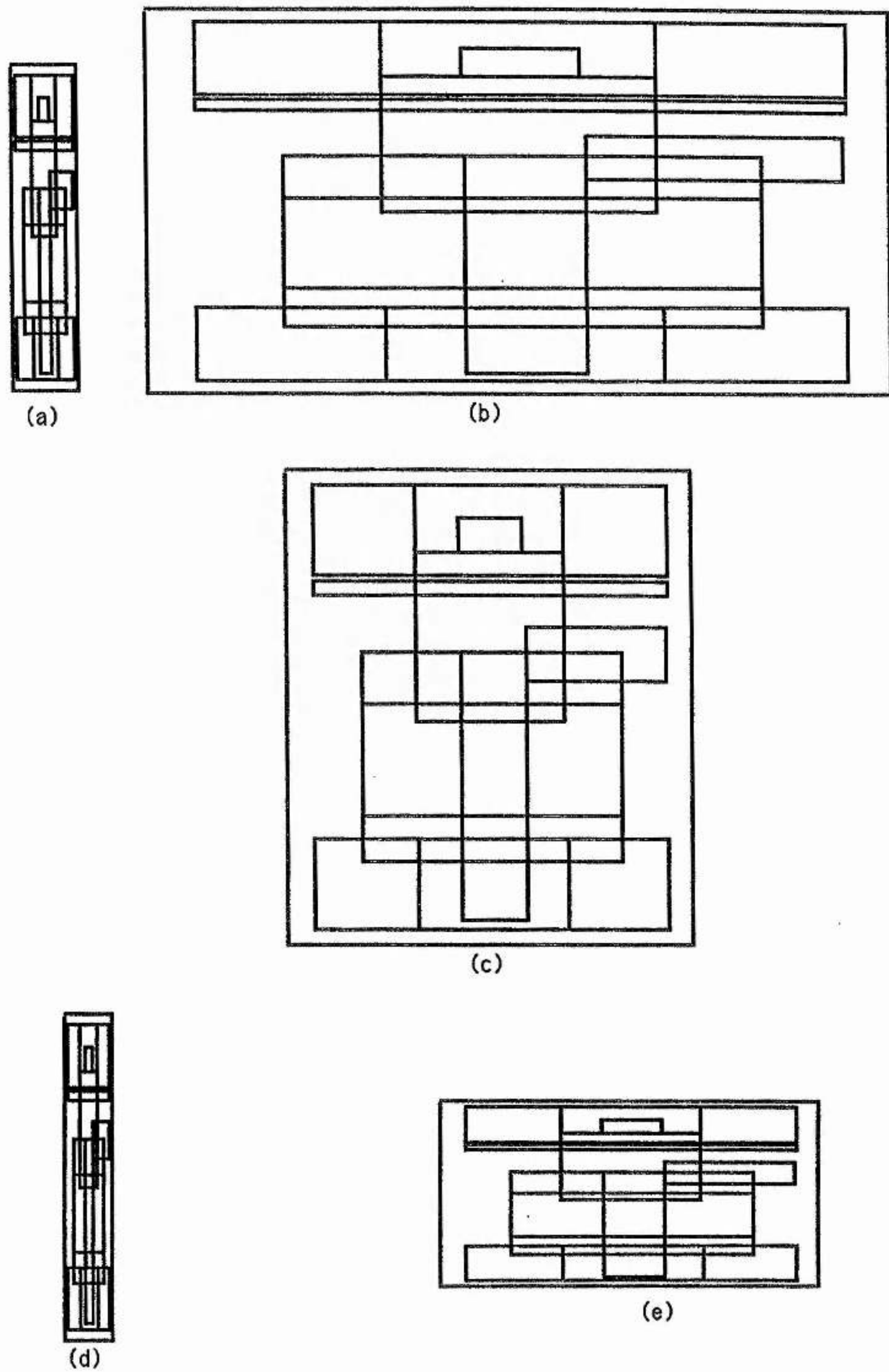


Figure 4.10 [contd]. Scaling effect on a VLSI design image.
Figure a,b,and c are scaled exactly without losing any information, but in Figure d, and e, some white pixels are turned to zero i.e dead short.

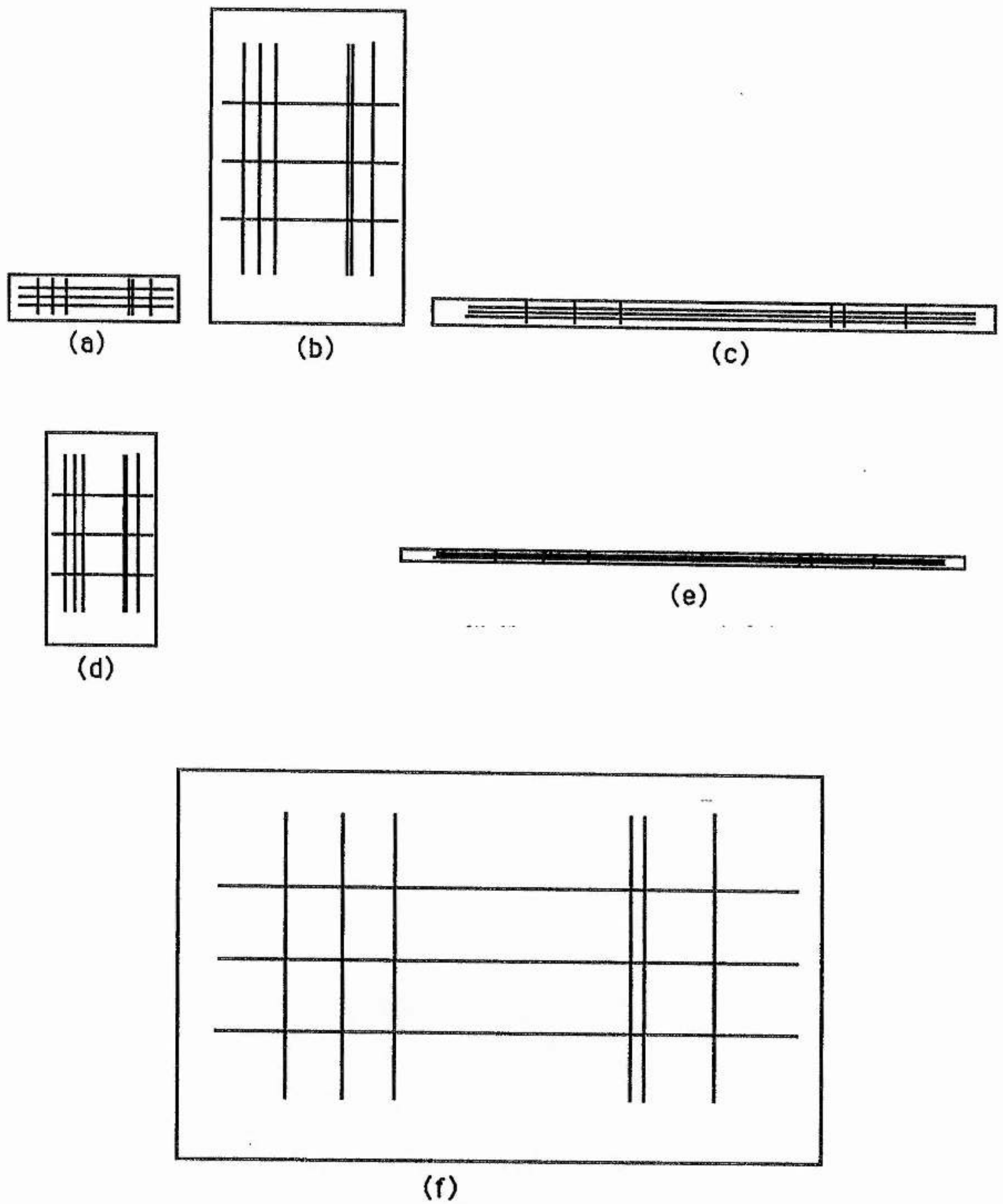


Figure 4.10[contd].

Figure a,b,c,and f showing the exact compression whereas in Figure d, and e there is a dead short i.e.,some of the white pixels are turned to zero.

4-4.3 Results :

An Image	Initial Radices $r_1 r_2 r_3 r_4 r_5 r_6$	Final Radices $r_1 r_2 r_3 r_4 r_5 r_6$	Scaling Factor	
			x-factor	y-factor
Image a	11 11 3 3 3 3	5 11 3 3 3 3 3 11 3 3 3 3 7 5 3 3 3 3	5/11 ---- 7/11	11/11 ----- 5/11
Image c	11 11 3 3 3 3	5 9 3 3 3 3 7 9 3 3 3 3 7 5 3 3 3 3	5/11 7/11 ----	9/11 9/11 -----
Image e	7 7 3 5 3 1	5 5 3 5 3 1 5 7 3 5 3 1 7 5 3 5 3 1 7 3 3 5 3 1	--- --- 7/7 ---	--- --- 5/7 ---
Image e	7 5 9 7	5 5 9 7 7 3 9 7 7 4 9 7 5 7 9 7	---- ---- 7/11 ----	---- ---- 4/11 -----
Image f	13 13 3 3 3 3	11 13 3 3 3 3 13 11 3 3 3 3	----- -----	----- -----
Image f	39 39 3 3	31 38 3 3 39 31 3 3 31 39 3 3	------ ------ 31/39	------ ----- 39/39

Table 4.4 Scaling of the images with different scaling factors.
'----' line indicates that the image is overscaled

The result shown in Table 3.4 is obtained before using the procedure put.1.for.0.black().

An Image	Initial Radices $r_1 r_2 r_3 r_4 r_5 r_6$	Final Radices $r_1 r_2 r_3 r_4 r_5 r_6$	Scaling Factor	
			x-factor	y-factor
Image a	11 11 3 3 3 3	5 11 3 3 3 3	5/11	11/11
		7 11 3 3 3 3	7/11	11/11
		3 11 3 3 3 3	3/11	11/11
		7 5 3 3 3 3	7/11	5/11
		7 3 3 3 3 3	7/11	3/11
Image e	7 7 3 5 3 1	5 5 3 5 3 1	5/7	5/7
		5 3 3 5 3 1	5/7	3/7
		7 4 3 5 3 1	7/7	4/7
Image e	7 5 9 7	5 5 3 5 3 1	5/7	5/5
		5 3 3 5 3 1	5/7	3/5
		7 3 3 5 3 1	7/7	3/7
Image c	11 11 3 3 3 3	7 9 3 3 3 3	7/11	9/11
		5 9 3 3 3 3	5/11	9/11
		7 7 3 3 3 3	7/11	7/11
		9 9 3 3 3 3	9/11	9/11
		9 10 3 3 3 3	9/11	10/11
Image f	13 13 3 3 3 3	11 11 3 3 3 3	11/13	11/13
		9 11 3 3 3 3	9/13	11/13
		7 11 3 3 3 3	7/13	11/13
Image f	39 39 3 3	31 31 3 3	31/39	31/39
		31 30 3 3	31/39	30/39
		25 35 3 3	25/39	35/39
		25 31 3 3	25/39	31/39
		21 35 3 3	21/39	35/39
		21 31 3 3	21/39	31/39
		19 37 3 3	19/39	37/39

Table 4.5 Scaling of the images with different scaling factors.

The result shown in Table 4.5 is obtained after using the procedure put.1.for.0.black.

The results obtained by scaling down the images are shown in Table 4.4 and Table 4.5. The results shown in Table 4.4 are obtained before using the procedure *put.1.for.0.black().* It should be noticed that if the surfaces in an image are wide spread from each other(e.g., chapter 3, Figure 3.2 a,b,and c) then the scaling factor before and after using the procedure *put.1.for.0.black* does not change very much. But in the case of wire frame images (e.g, VLSI design , chapter 3, Figure 3.2 e,f) there is too much variation in the x and the y scaling factors. The result shown in Table 3.5 is obtained after using the procedure *put.1.for.0.black().* From the results shown in Table 4.4 and 4.5, the difference between the permissible x-scaling factor (or the y-scaling factor) for the images e and f is large. In the case of image e with six radices (see Table 4.4) the minimum value for the x-scaling factor is 7/7 and for the y-scaling factor is 5/7, whereas as shown in Table 4.5 the same image has 5/7 and 3/7 as the minimum x and y-scaling factors(i.e., 20 pixel less than the previous tile). Similarly for the image e with 4 radices. From Table 4.4 it has shown that the image f with six radices cannot be scaled down whereas in Table 4.5 the minimum scaling factors for x and y are 7/13 and 11/13 (i.e. 92 pixels less then the previous one).

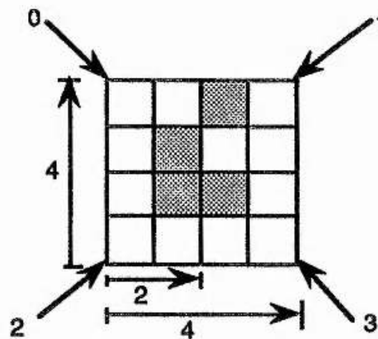
From the above result and the discussion, we can now say that the procedure *put.1.for.0.black* helps in approximately compressing the images. We are not considering the cases of scaling up the images since turning of black pixels to zero is impossible. Scaling up the images works with any scaling factor provided the quotient for the x-factor is not even.

4-5 Remarks :

The results obtained for converting scan1 into scan2 using a standard murray scan were compared with those obtained by using linear murray scan. In most cases a linear murray scan takes less time than that of the general

murray scan (see Table 4.3). But as discussed earlier that the distribution of runs obtained by general murray scans will be different to that of standard linear scan with fly back, which may result in better compression.The same argument can be applied to show the advantage of general murray scans over linear murray scans.

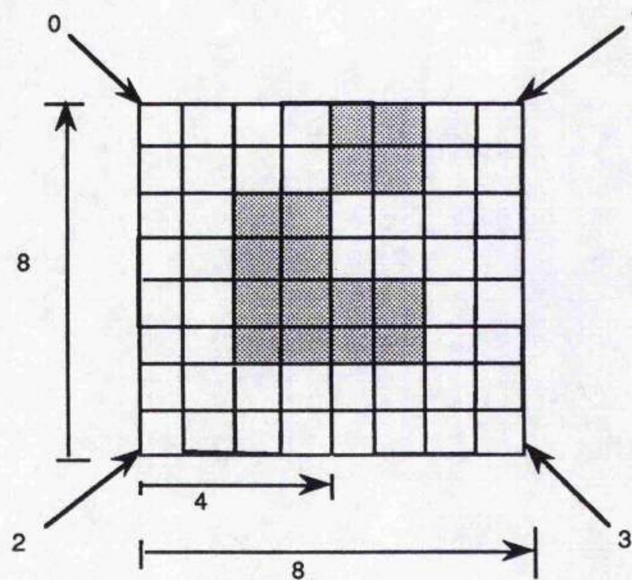
Further in the case of quadtrees the scaling factor depends upon the size of the image i.e., if the initial size is 4 then the next size of the image can be 8 or 16 or 2 etcetera, which a disadvantage of this method. As we have seen in the case of murray scan the scaling factor is independent of the size of the image. But if the scaling factor is greater than or equal to 2 then quadtree methods will be more effective than that of murray scans, since in case of quadtrees the same codes can be used to regenerate the scaled image. For example, an image of size 4*4 with some black pixel is given below.



The corresponding codes in the ascending order for these black cells will be ,

30, 21, 10, 03

Now if we scale the image with a scaling factor equal to 8/4 then using the same codes we can generate the scaled image see below.



From the above example we can say that the quadtree approach is efficient, but it depends upon the size of the image. The next scaling factor for the above example will be 16/8. For the quadtree approach we cannot use any other scaling factor between 8/8 and 16/8, whereas in the case of the murray scan we can have more number of scaling factors between 8/8 and 16/8 giving the exact scaled image, which is an advantage of the murray approach over the quadtree approach.

Chapter 5

5. SUPERIMPOSITION, AND SET OPERATIONS ON IMAGES 137

5-1 Introduction 137

5 - 2 Set Operations 138

5-3 Superimposition of The Images Using Murray Polygons 140

Implementation 141

5 - 4 Set Operations Using Murray Polygons 143

Union 144

Intersection 146

Difference 146

5 - 4 Remarks 147

5-1 Introduction :

If we consider a complex image we often find that this is made up of many simpler sub-images, some of which may be transforms of each other. For example, an image of a busy intersection has cars of different sizes in different locations and moving in different directions. In addition there are people, street signs, trees, etcetera. With time there will be some changes in the image. The change may come to the number of cars, and people at the intersection. Hence we should have the capability of changing parts of an image, while keeping other parts fixed.

Set operations on the images can be used to merge two or more images together. Medical 3D.images which are obtained by putting planes one after other can be reduced to a single plane i.e., 2-dimension, by merging all the planes together, thus removing the hidden surface area from a given view point. The hidden surface problem is discussed in detail in the following chapters. Also in Constructive Solid Geometry(refer Hearn, and Baker(1986)) solids(such as spheres, cubes, cylinders, etcetera) can be combined by geometrical transformation and boolean operations such as union,intersection and difference. Union of the images is similar to the '*or*' operator, whereas intersection and difference of the images is similar to '*and*' and '*xor*' operators respectively.

In this chapter, initially past and present work is categorised and briefly discussed. Superimposition of images and some set operations using murray techniques is discussed. All the operations are on the runlengths only, which are stored either in a file or in a list. We do not need to go back to the image once we have got the collection of runlengths corresponding to that image.

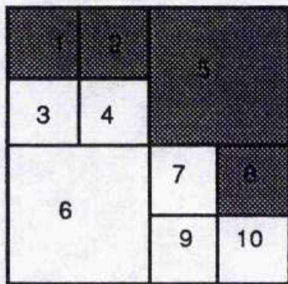
5-2 Set Operation :

The quadtree is especially useful for performing set operations such as union and intersection of several images. This is described in greater detail by Hunter and Steiglitz(1979a), and Shneier(1981). The union of the quadtrees, S and T can be obtained by examining the corresponding nodes and constructing the resulting quadtree, say in U. If either of the two nodes is BLACK, then the corresponding node in U is BLACK. If one node is WHITE, say in S, then the corresponding node in U will corresponds to the node in T. If both nodes are GRAY, then U is set to grey and the algorithm is applied recursively to the sons of S and T. Once the sons have been processed, a check will be made to see whether a merger is to take place, since all four sons could be BLACK. For example, consider the union of the quadtree of Fig 5.1 and 5.2. Node B in Fig 5.1 and node E in Fig 5.2 are both GRAY. However the union of their corresponding sons yields four BLACK nodes in U. Fig 5.3 shows the union of Fig 5.1 and 5.2.

Computing the intersection of two quadtrees is just as simple. The algorithm described above for union is applied, except that the roles of BLACK and WHITE are interchanged. If either of the two nodes is BLACK, then the corresponding node in U is BLACK. If one node is WHITE, say in S, then the corresponding node in U is WHITE. The check for a merger is performed to determine if all four sons are WHITE. Figure 5.4 shows the result of the intersection of Figure 5.1 and 5.2.

Gargantini(1982) used linear quadtree for union and intersection of images. Here for union, if two nodes are the same, then only one node is stored. If one block covers the other one then the larger block will be stored in the final array. Intersection can be treated similarly. For example, let

C1 = { 003, 021, 023, 03X, 122, 21X, 3XX },



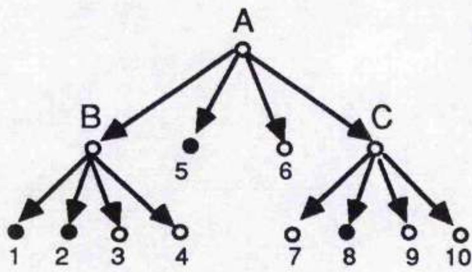


Figure 5.1. Sample image and its quadtree.

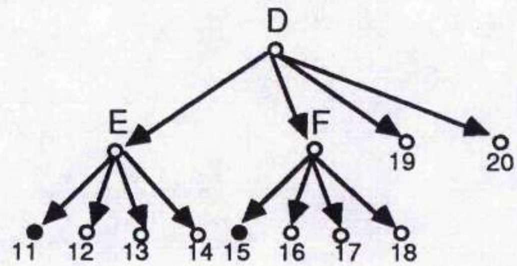
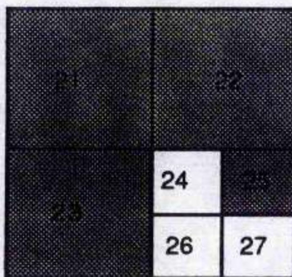


Figure 5.2. Sample image and its quadtree.



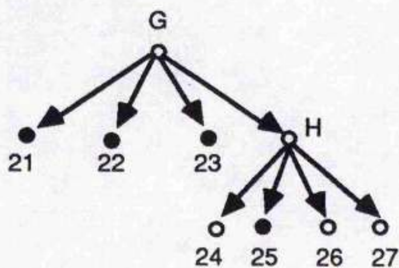
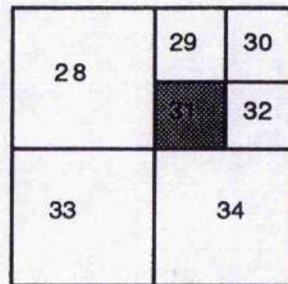


Figure 5.3. Union of the images in Figure 5.1 and 5.2.



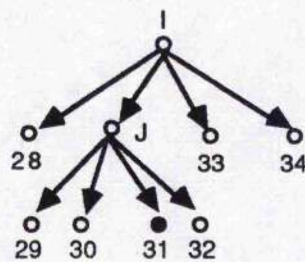


Figure 5.4. Intersection of the images in Figure 5.1, and Figure 5.2.

and

C2 = { 122, 111, 113, 131, 133, 230, 231, 3XX } be the linear coding for the given two images. If we find the union of the two images, we obtain,

C1 U C2 = { 003, 021, 023, 03X, 111, 113, 122, 131, 133, 21X, 230, 231, 3XX }.

Gargantini(1983) used the same process for the linear octree .

Burton, Kollias, and Kollias(1987) defined a simple map overlay function for quadtrees, and showed that many common quadtree operations including union and intersection, can all be considered as special cases of the map overlay function. The process to overlay different types of site data to produce some kind of composite map is commonly referred to as the map overlay problem. For example, a quadtree representation for two maps partitioned into districts in two different ways is given. One correspond to a soil map and the other to political map. A district within a map may correspond to a state defined area (e.g., country), to a soil type (e.g., Alfisol), or to an elevation range. Here they use integer numbers for representing the different districts. The overlay map for each (soil type, political unit) pair is obtained by multiplying the number of the first district by a factor (say 10) , and adding the number of the second district. The new district with number 23 (say) would be the intersection of (soil) district 2 with (political) district 3. Union , Intersection and Difference can apply only to 0-1 quadtrees, where 1 indicates that the (soil type, Political unit) pair satisfies some particular conditions. More detail can be found in Burton, Kollias, and Kollias[1987].

Oliver and Wiseman(1984) used a treecode representation for merging two or more images. Treecode representation is defined in section 1-3.2. Merging is done by examining the node of each input tree. If at a certain stage the next node in each tree is a leaf node then the leaves are combined to form an output node. If the first tree contains a non-terminal node and the second a leaf, then the first scan recurses until it catches up

with the second, and vice versa. If both trees contain non-terminal nodes both scans will recurse.

5-3 Superimposition Of The Images Using Murray Polygons :

One way of superimposing one image on top of another is to copy the changed part on to the image. This expression will persist so long as it is on the screen. There will be no change in the runlengths. Next time to obtain this changed plane, we have to do the same calculations again. This is not convenient for an animated movie where we have to run, say 20 frames a second to get the moving expression. Another way of superimposing one image on top of another is to copy the changed part in such a way that the previous runlengths will change to give the sequence of runlengths for the new plane.

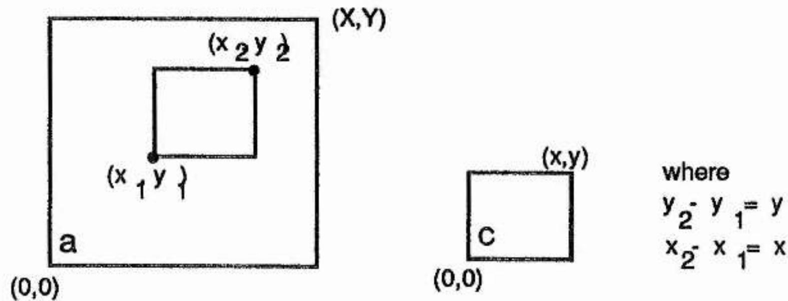
Here we present a algorithm which has the capability of superimposing one image on top of another by changing the runlengths. The only information which is required is the size of the image which is going to be superimposed and the starting point on the image where it is going to be superimposed. The starting point can be obtained with the help of the function *'Locator'* which gives the position of the mouse relative to the screen. The size of the image is required to decide about the x and the y-radices.

Both the images (say A and B) are stored in the database, whereas image A is the initial image and image B is going to be superimposed on image A. Initially image A will be scanned using a murray scan and the corresponding sequence of runlengths will then be stored either in the database or in a file. The algorithm will now take image B and the runlengths corresponding to image A as an input. Output will be the collection of runlengths corresponding to image A with image B superimposed on it. We will discuss this in the next section. Once the desired runlengths have been obtained, the image can be

drawn on the screen. The data structure used to store the runlengths and the implementation part is discussed in the next section.

5-3.1 Implementation :

Consider two images of different sizes as shown,



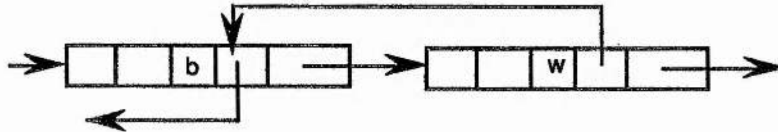
where 'a' is the initial image and 'c' is the image which has to be superimposed on the image 'a'. The position where the image has to be superimposed has been marked. Let (x_1,y_1) be the starting point in an image 'a' for the image 'c' to be superimposed. Our first step is to scan the image 'a' to give the sequence of runlengths. The image 'a' can be removed once we have obtained the corresponding runlengths. The image 'c' can similarly be removed once scanned, and the corresponding runlengths can be used to superimpose 'c' on 'a'. But the scanning part and the superimposition part, of 'c' on 'a' can be done simultaneously, which is discussed later. Time can be saved by not rescanning the image 'c'. The data structure used for the list to store the runlengths is same as used before. We define our data structure as,

structure integer.list(**int** runlength, Sum;**string** colour;**pntr** left,next)

The linked list obtained after scanning the image 'a' uses the above *structure* to store the runlengths. Each cell contains information about the

consecutive pixels of the same colour. All the items used in the data structure are discussed in chapter 4 .

Let $r_1, r_2, \ldots\ldots\ldots\ldots\ldots, r_n$ be the runlengths corresponding to the image 'a' . Each runlength corresponds to a colour. The colour information for each runlength is given in the item *'colour'* defined in the structure. Since we are considering black and white images, the item *'colour'* contains either 'w' i.e., white or 'b' i.e. black, as the two values. The linked list is as shown,



Consider the image 'c' which has to be superimposed on image 'a'. We will scan this image using a murray scan. The first point on this image is (x,y) = (0,0). But since the starting point for the image 'c' on the image 'a' is (x_1,y_1) , we will shift this point by (x_1,y_1) i.e., the new values for the x-coordinates and the y-coordinates will be given as,

$$x(final) = x + x_1 = 0 + x_1 = x_1$$

$$y(final) = y + y_1 = 0 + y_1 = y_1$$

Using the transformation f(x,y) ---> n (discussed earlier), we can find the corresponding nth point on the image 'a'. Now scan the list obtained for image 'a' and consider that cell where this nth point belongs. The colour on both the images will now be compared.

If the colour corresponding to the initial point (x,y) (= (0,0), say) in image 'c' and the colour corresponding to the final point (x,y) in an image 'a' is black then there will be no change in the list.

If the colour corresponding to a point in image 'c' is black and the colour corresponding to the point (x,y) in an image 'a' is white then we will change the runlengths by turning that pixel to black.

If the colour corresponding to a point in image 'c' is white then there is no need to consider that point.

For all the points in an image 'c', which are black we will repeat the above scheme. The output will be another frame (or image) with the slight change, in comparison to the previous one. This method is similar to finding a union between two images. Set operations are discussed in the next section.

5.4 Set Operations Using Murray Polygons :

The set operations which are considered in this section are,

i. Union,

ii. Intersection,

iii. Difference.

If A and B are two images which are black and white , then we define,

$A \cup B$ = the set of all the black pixels which are in A as well as in image B

$A \cap B$ = the set of all the black pixels which are common in image A and in image B.

$$A \Delta B = (A \cup B) - (A \cap B),$$

i.e. the set of all the black pixels which are in A and in B except those which are common to both the images.

The images can be merged either by using the images themself or by using the different sets of runlengths obtained by scanning the different images. In the first case, the whole image must be in the system while processing. This will require a large amount of memory for images having a few thousand surfaces(or planes). Secondly it can be time consuming, since we have to compare each pixel of an image to the other one. A second approach can be more efficient than the first one. We do not require all the planes at the same time. Here we scan the planes one by one to get the sequences of runlengths, and then merge them together to give a single sequence of runlengths. The points to be remembered are that the size of all the images should be same. The x-radices and the y-radices can be same or different for all the images. This is discussed later on.

5-4.1 Union :

Let $r = r_1, r_2, \ldots\ldots\ldots\ldots, r_n$ and $s = s_1, s_2, \ldots\ldots\ldots\ldots, s_m$ be the collection of runlengths corresponding to the two images which we have to merge. The radices used to scan the images should be same. Here our problem is to obtain the sequence $t = t_1, t_2, \ldots\ldots\ldots\ldots, t_k$ such that the sequence t has black runlengths where either the sequence r or the sequence s or the both sequences have black runlengths.

Now to obtain the sequence t we will traverse both the sequences r and s together. Three cases are possible between the two sequences r and s.They are,

1. $s_i = r_j$. Output the one i.e. s_i or r_j, which is black otherwise any of the two.

2. $s_i > r_j$. There can be two cases,

   i. s_i is black.

      output s_i with the associated colour and then scan the sequence r, until the sum of the runlengths starting from r_j is greater than or equal to s_i.

      Define $sum = r_j + r_{j+1} + \ldots\ldots\ldots + r_k >= s_i$. We can consider two cases ,

      1. $sum = s_i$. Compare the next runlength of the two sequences i.e r_{k+1} and s_{i+1}.

      2. $sum > s_i$. Since the runlength s_i has already been output in the final list, we will compare the remaining runlength $sum - s_i$, which belongs to the member r_k of the sequence r, with the next runlength of the sequence s i.e. s_{i+1} .

   ii. s_i is white.

      scan the sequence r, until the sum of the runlengths starting from r_j is greater than or equal to s_i. Define $sum = r_j + r_{j+1} + \ldots\ldots\ldots + r_k >= s_i$. Here we will output $r_j, r_{j+1}, \ldots\ldots\ldots, r_{k-1}$ with their associated colour. We consider two cases ,

      1. $sum = s_i$. Output r_k with the associated colour and then compare the next runlength of the two sequences, i.e r_{k+1} and s_{i+1}.

      2. $sum > s_i$. Output $r_k - (sum - s_i)$ with the colour corresponding to the member r_k of the sequence r and then compare the remaining runlength $sum - s_i$, which belongs to the member r_k of the sequence r, with the next runlength of the sequences i.e. s_{i+1} .

3. $r_i > s_j$. This is similar to the case 2, only change being to replace r by s and s by r.

5-4.2 Intersection And Difference:

Let $r = r_1, r_2, \ldots\ldots, r_n$ and $s = s_1, s_2, \ldots\ldots, s_m$ be the collection of runlengths corresponding to the two images which we have to merge in order to get the common pixels. The problem is to obtain the sequence $t = t_1, t_2, \ldots\ldots, t_k$ such that the sequence t has black runlengths where both sequences r and s have black runlengths, and if either r or s is black then the sequence t will assume one which is white. If the radices used to scan the images are same, the above method discussed for the union case can be slightly modified to get the sequence t which will remove all the areas which are not common to both the images. Here instead of modifying the above method we will discuss another approach to this problem. Here the radices used to scan the images can be same or different.

Scan the sequence $s = s_1, s_2, \ldots\ldots, s_m$ and consider each member of this sequence in order. Using the transformation (n ----> f(x,y)) explained in chapter 2, we can find the corresponding coordinates for each point in the sequence s. For example,

> If $s_1 = 3$ and $s_2 = 4$ then the corresponding values for n, in the cell s_1 will be 0,1, and 2 and in cell s_2 they will be 3,4,5, and 6. For any cell s_k the corresponding values can be obtained by using the item *'Sum'* defined in the structure above. The first point for the cell s_k will be equal to the *'Sum'* value plus one.

Once we have the co-ordinates, using the transformation (f(x,y) ---> m) we can find the corresponding mth point in the sequence r. Now we have to

compare the colour for the nth point in the sequence s with the mth point in the sequence r.

> If both are black or both are white then there will be no change in the sequence r.

> If one is black and other one is white then the colour of the mth point in the sequence r will change to white if it is black.

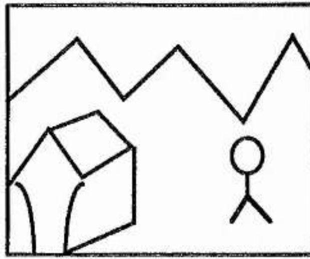
The new sequence r will be the required sequence t. Similarly if we have to find the *difference* between the two or more images then the colour comparison for the two points in the two sequence will be slightly different. Here,

> If both are black or both are white then the colour of the mth point in the sequence r will change to white.

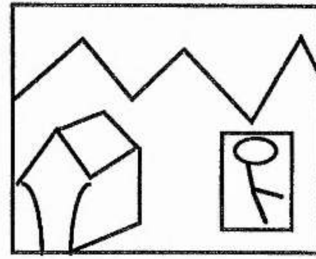
> If one is black and other one is white then the colour of the mth point in the sequence r will change to black if it is white.

5-5 Remarks:

Superimposition of one image on top of other may be helpful in the case of an animated movie. For example, if we see two consecutive planes of an animated movie as shown below, the only change which we can see is the position of the character which has moved slightly. Therefore we need to change only the part which is given in the small rectangle keeping the surrounding objects the same.



(a)



(b)

Since the character is moving, we have to consider the changes in with the character and with the background portion, which is changing with respect to the character*(Note : we consider the background scenery as stationary).* To get the consecutive planes we can simply modify the background plane. Initially draw the background image and then superimpose the moving character at a given point.

Chapter 6

6. CONNECTED COMPONENET LABELLING 149

6 - 1 Introduction 149

6 - 2 Connected Component Labelling 151

6 - 3 Connected Component Labelling Using Murray Polygons 153

Method 1(Using Images) 155

Method 2 162

Using Two Sequences of Runlengths 162

Extension To 3-Dimensional and n-D Images 168

Comparison Between Method 1 And Method 2(part 1) 170

Using One Sequences of Runlengths 170

Extension To 3-Dimensional and n-D Images 178

Comparison Between Method 2 (Two list vs One list) 179

6 - 3 Remarks 179

6.1 Introduction :

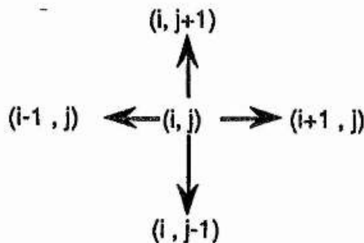
Connected component labelling [refer Rosenfeld and Kak(1976)], is one of the basic operations of an image processing system. It is analogous to finding the connected components of a graph. Let S be a finite set of pixels. Two pixels (i,j), (K,J) ε S are n-connected in S if and only if there is a n-path between (i,j) and (K,J) consisting entirely of points of S i.e., a sequence of elements, $(i,j) = (i_0, j_0), (i_1, j_1), \ldots\ldots, (i_n, j_n) = (K,J)$, all in S such that (i_r, j_r) is a neighbour of (i_{r-1}, j_{r-1}), $1 <= r <= n$. Connectedness may be defined in terms of the neighbours of a point (i,j). Let (i,j) be a point of the given image. Then (i,j) has four horizontal and vertical neighbours, namely the point

$$(i-1,j), \quad (i,j-1), \quad (i,j+1), \quad (i+1,j)$$

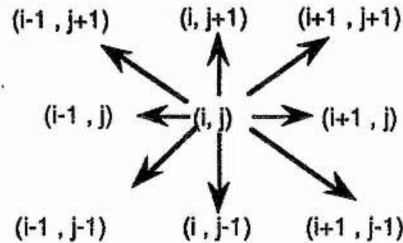
These points are called the 4-neighbours of (i,j), and are said to be 4-adjacent to (i,j). In addition, (i,j) has four diagonal neighbours namely

$$(i-1,j-1), \quad (i-1,j+1), \quad (i+1,j-1), \quad (i+1,j+1)$$

Both these and the 4-neighbours are called 8-neighbours of (i,j), as shown,



4-point connectivity



8-point connectivity

Finding connected components can be very useful in some areas of image processing. Consider for example a medical image which shows a tumour surrounded by other parts of the body (e.g. bones, lungs, etcetera). A doctor cannot have a better look of the tumour since it is surrounded by other parts. It will be better if we can extract the tumour out leaving behind the parts obscuring it. This we can do by finding the connectivity between different parts, supposing that the tumour has some physical property to distinguish it from the surrounding tissue. Those parts which are not connected to that can be removed. Similarly for a underwater picture. Here for example an oil pump which is surrounded by fish, algae and other materials. One can remove all the unwanted substances (such as algae, fish etcetera) from an image, leaving behind the one in which we are interested. In many other places also connectivity can similarly be applied.

To some extent connectivity can be helpful in compressing the data approximately. The initial sequence of runlengths which has been obtained after scanning an image will contain all the black chunks which are present in an image. Using connected component labelling we can find the required connected component in an image and can remove those components which are not required. The runlengths corresponding to those parts which are removed from the scene will be merged with the back ground color i.e., will turn to white, resulting in a smaller number of runlengths. For example,

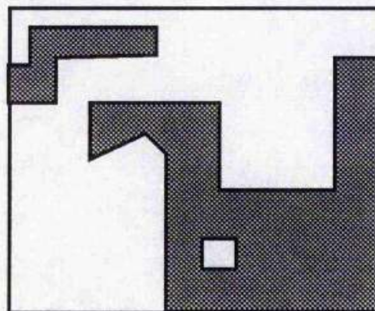
w	b	w		w	
5	7	3	----->	15	*the black runlength has merged with the two adjacent white.*

In this chapter, initially past and present work is categorized and briefly discussed. Different algorithms using murray polygons are presented to find the connected components of an image. Different images which are

given in chapter 3, Figure 3.2 are considered by these algorithms. Completion time for different images, using different algorithms which use murray techniques have been compared. Comparison has also been done between algorithms which use murray techniques and with those obtained with quadtree, octree, or linear encoding. All the images which are considered are black and white.

6.2 Connected Component Labelling :

As defined above, connected component labelling is one of the basic operations of an image processing system. For example, the image shown below has two components. Given a binary array representation of an image, Rosenfeld and Pfaltz(1966) suggested a "*breadth-first*" approach, which scans the image row by row from left to right and assigns the same label to the adjacent BLACK pixels that are found to the right and in the downward direction. During this process pairs of equivalences may be generated, resulting in two more steps. The first step is to merge the equivalences and the second one is to update the labels associated with the various pixels to reflect the merger of the equivalences.



An image with two connected blocks. Blocks in the image are shaded; background blocks are blank (white).

Samet(1981a) used a quadtree method to perform the same operation. His algorithm is three-step process. The first step is a postorder traversal (in order NW,NE,SW,SE), where for each BLACK node that is encountered, say A, all adjacent BLACK nodes on southern and eastern sides of A are found, and assigned the same label. The adjacency exploration is done using the neighbour-finding techniques of Samet(1982). The second step merges all the equivalence pairs that were generated during the first step. The third step performs another traversal of the quadtree and updates the labels on the nodes to reflect the equivalences generated by the first two steps of the algorithms.

Gargantini(1982) used linear quadtrees and showed how to find the pixel adjacent to a given one in a specified direction. If $K = (k_{n-1}K_{n-2}...........K_0)_4$ be the given pixel and $S = (S_{n-1}S_{n-2}.............S_0)_4$ its adjacent node in a particular direction (say southern direction) then the problem is how to determine digits $S_{n-1}, S_{n-2}, ..., S_0$. Here two case are distinguished: in the first one, K and S belong to the same quadrant relative to the nth subdivision; in the other, K and S do not. Four different algorithms are required to find the adjacent nodes in four direction (N, S, E, W). They have been explained (refer Gargantini(1982)).

Unnikrishan(1987) proposed a connected components algorithm using a linear hierarchical quadtree (LHQT). LHQT is obtained by rearranging a linear quadtree into a hierarchy of arrays based on the size of the black node, as defined earlier. The algorithm explores adjacencies in the LHQT, assigning unique labels. It has shown that the use of the LHQT in connected component labelling results in greater computational efficiency than the algorithm given by Gargantini(1982).

Gargantini(1983) used a linear octree to find the adjacent pixels in a three-dimensional image. His algorithm produces the pixel adjacent to an

internal one in a specific direction. A binary search can be used to determine whether or not the found pixel is black as in the planar case Let Q be represented by an octal digit $q_{n-1}, q_{n-2}, \ldots\ldots\ldots q_0$. Here for each digit two cases have been considered to find the octal code of the pixel adjacent to Q in the Eastern direction (say).

1. If q0 is even, the adjacent node in the Eastern direction belongs to the same octant as Q and therefore $E(q_0) = q_0+1$, $E(q_j) = q_j$, $j = 1, 2 \ldots\ldots n-1$.

2. If q_0 is odd, Q and its adjacent node belong to two different octants. The adjacent octant is represented by (q_0+7) rem 8. The other digits are determined by analyzing $q_1, q_2 \ldots\ldots$, if even then we use case 1 otherwise case 2.

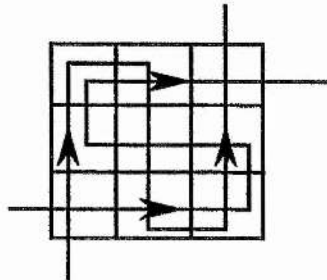
For the boundary pixels, if the found value of E(Q) is such that $Q-E(Q) = (11\ldots\ldots1)_8$ then Q is on the eastern border. If $Q-S(Q) = (22\ldots\ldots2)_8$ then Q is on he southern border. Similarly internal or boundary pixels can be determined for other directions also.

6.3 Connected Component Labelling using Murray Polygons :

As defined above, connected component labelling is the process of identifying the disjoint elements of the image. If a binary array representation of an image is given, the simple method of finding connectivity would be to scan the image row by row from left to right and assign the same label to the adjacent black pixels that are found to the right and the downward direction. This process is one dimensional and secondly all the pixels need to be considered in order to find the connectivity to the right and in the downward direction. Once the end point of an image has been reached the image will be rescanned from top to bottom. Rescanning is continued until the

connected component is obtained. The time required depends upon the number of black pixels and the shape of the component.

In the next sections we will discuss two different approaches for finding the connected components of an image. The efficiency for both methods is compared for different images. The theory behind these methods is very simple. Let us consider two perpendicular lines intersecting at a common point A (say). The common point A is connected with the four neighbour points of the two intersecting lines. Also we know that since the connectivity relation is transitive and reflexive then c R a if a R b and c R b . Since we have a common point and the relation is transitive it implies all the points are connected to each other. In the case of the murray scan this can be obtained by considering two different scans, one will subdivide an image array into horizontal tiles and the other one will divide the image into vertical tiles. For example, a tile of size 3*3 and the two scanning patterns corresponding to an image are as shown,



At any stage the two scans in a tile will be perpendicular to each other. The connectivity can now be obtained by using both the scans together. To find

connectivity we can start from a tile and from this tile we can find other tiles and so on, more detail follows.

6.3.1 Method 1 (*Using Images*):

Consider a black and white image with many connected areas as shown in Figure 6.1. Suppose we are interested in the blob marked T. Our problem is to extract this blob from the image leaving behind the other small ones.

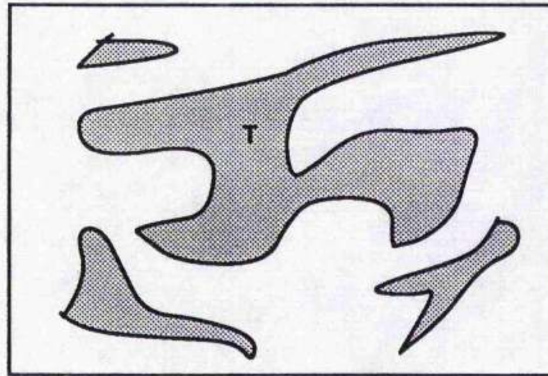


Figure 6.1. An image with many connected areas.

The approach is very simple. Initially scan an image horizontally and vertically to get the sequence of runlengths where the radices for the two scans can be the same or different. For the scanning part we can either use two scans separately which will break an image into collection of tiles scanned in the horizontal and in the vertical direction to give two sets of runlengths, or we can scan the image in the horizontal direction and then using the scan conversion algorithms discussed in chapter 4, the runlengths for the vertical scan can be obtained. The use of the scan conversion algorithm is to get rid of the image once scanned. The operations will then be on the runlengths only. But in comparison, both the methods approximately take the

same time to get the corresponding runlengths for the other scan. Here in this method we use two different scans to scan an image.

The starting point should be the subset of the set containing all the points belonging to the blob in which we are interested. A starting point in an image can either be indicated by positioning a cursor on the display or in an automated system choosing a long runlength of the appropriate colour only if the component required is large. In the later case a problem can arise when the image has two or three blobs of the same size. The starting point may belong to other blobs in which we are not interested. We have to repeat the process until we get the required blob. In this method we will choose a long runlength of black colour (*since the images used are black and white only*) to find a starting point. How to use cursor for finding the starting point is discussed in the second method. The complete method for finding the connected components by using images is as follows.

Let $r_1, r_2, \ldots\ldots, r_n$ -------------------(1)

be the sequence of runlengths obtained from the horizontal murray scan and

$s_1, s_2, \ldots\ldots, s_m$ ---------------------(2), be the sequence of runlengths obtained from the vertical murray scan. We can start with any of these sequences. Consider the first sequence of runlengths and find the maximum black runlength for the starting point. Let the maximum black runlength in the first sequence be r_i. Convert this black runlength to white by merging the two white neighbours with it. Now we have a new sequence of runlengths where the maximum black runlength has been changed to white. The new sequence is thus,

$r_1, r_2, \ldots\ldots, r_{i-2}, r'_i, r_{i+2}, \ldots\ldots, r_n$ ------------ (A)

where $r'_i = r_{i-1} + r_i + r_{i+1}$

Using this new sequence of runlengths an image can be drawn at any point on the screen. Scan this new image using a vertical murray scan.

Let S_1 , S_2 , ..., S_M --------------------(3), be the new vertical runlengths. Now we have two sequences of runlengths obtained by using the vertical murray scan.The first one is obtained by scanning the original image and the second one is obtained by scanning the new image. The two vertical sequences are represented by label (2) and (3). If we compare both the sequences we will find that some of the black runlengths are identical, whereas some of them are changed. The black runlength in the sequence (2) which has changed in (3) are actually broken into several runlengths. The reason is the black area which has changed to white. All the black runlengths in the sequence (2) which pass through that area will be affected. For example, suppose s_i is the only black which has changed. All the runlengths before and after it will be same , see below.

w b b w b/w

s_1 ,s_2 , ..., s_i , s_{i+1}, ..., s_m

S_1, S_2, ..., S_i , S_{i+1}, ..., S_{i+k} , ..., S_M

where $s_i = S_i + S_{i+1} + \dots + S_{i+k}$,

$s_i = S_i$ for $i = 1$ to $i-1$,

and $s_i = S_{i+k}$ for $i = i+1$ to m.

Here s_i has broken into k runlengths. Since s_i is black and so the runlengths S_i , S_{i+2},............, the relation is transitive implies all these black runlengths are connected to each other. Our next step is to convert s_i to white as we did earlier. The new sequence of runlengths for the vertical scan is now,

w b b w b b/w

$s_1, s_2, \ldots\ldots, s_{i-2}, s'_i, s_{i+2} \ldots\ldots, s_m$ ----------- (B)

where $s'_i = s_{i-1} + s_i + s_{i+1}$

Again the image can be drawn on the screen using the runlengths given in (B) . This time we will use the horizontal murray scan to scan this new image. The new sequence of runlengths will now be compared with the sequence given in (A) and so on. When there is no change in the two sequences which we are comparing our program will stop. The whole scheme is given in Figure 6.2.

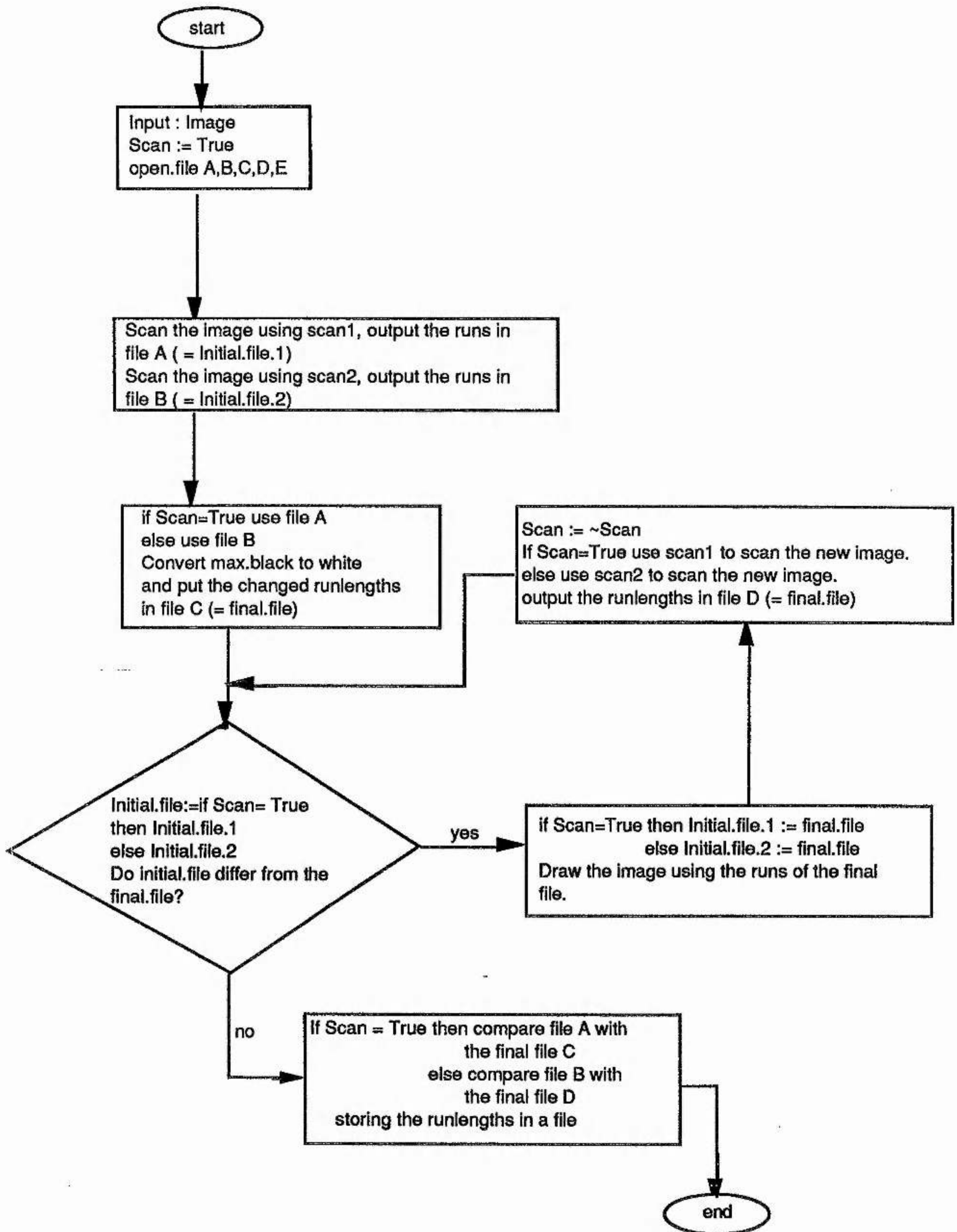


Figure 6.2. Flow chart giving the scheme for method 1.

Now the final runlengths will have the blob marked *T* totally removed i.e., turned to white. To get the runlengths for the blob, we may either accumulate them as we find them or compare the final horizontal sequence with the initial sequence given in (1). Those pixels which are black in the final sequence will have turned to white in the sequence given in (1), resulting in the sequence of runlengths for the blob marked *T*. This is discussed below,

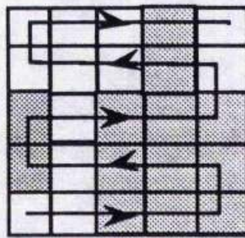
Let $r_1, r_2, \ldots\ldots, r_n$ be the initial runlengths given in (1) and

let $r'_1, r'_2, \ldots\ldots, r'_m$ be the final runlengths where $n>m$. To get the runlengths for the blob we will scan both the sequences together to look for the black common pixels in the two sequences. A runlength r_i is said to be common to the runlength r'_j of other sequence if,

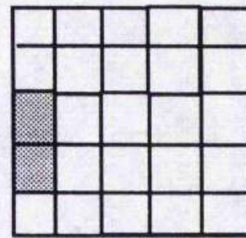
$r_1+r_2+\ldots\ldots+r_{i-1} = r'_1+r'_2+\ldots\ldots+r'_{j-1}$, where r_i and r'_j are both black.

Therefore to scan both the sequences we need to store the information corresponding to the left sum i.e., how many pixels have been used before a runlength (say r_i). We can define *sum1* and *sum2* to be the initial sum for the two sequences, where initial values for *sum1* is equal to r_1 and for *sum2* equal to r'_1. Whenever we encounter a new runlength we will add this to the previous sum to give the new sum value. Now the movement of each sequences depends upon the relation between the two sums. Two case need to be considered, *sum1* < *sum2* or *sum1* = *sum2* *(Note : sum1 cannot be greater than sum2 since the runlengths r'_j will always be greater than or equal to r_i).*

If *sum1* < *sum2* this implies that $r'_j > r_i$. Since both the sequences are obtained by using the horizontal murray scan, we will conclude that the runlength r'_j is obtained by adding some of the runlengths belonging to the blob, see below,



(a)



(b)

Here (a) is the initial image and image (b) is the final image obtained by turning the biggest blob to white. The two sequences of runlengths obtained from the images (a) and (b) are,

		w b w b w b w b w b w
Initial	:	2 6 1 2 1 3 1 1 6 1 1
Final	:	9 2 14

If we see the final sequence then the runlength 9 is equal to the sum of the first three runlengths in the initial sequence, and similarly for the runlength 14. Therefore in these cases the first sequence will scan the consecutive runlengths storing them in a file, while the second sequence waits for the first to catch up i.e., $sum1 = sum2$.

If $sum1 = sum2$, this case will arise when the scan touches other surrounding blobs which are not connected to the blob in which we are interested. Here in the first sequence we will change the next runlength to white if it is black, since the next runlength does not belong to that blob. For example if we consider the same sequences of runlengths given above we will see when both the sums are 9 then the next runlength i.e., 2 belongs to the other blob. Keep on repeating this until the end of the two sequences is reached.

The same idea for 2-D images can be further extended to 3-D and n-dimensional images. But since we have to draw and scan an image whenever a new sequence of runlengths has obtained after comparison, the time of completion for this process will be very high. Since scanning and drawing part of an image is very time consuming, methods other than murray polygons (i.e. linear encoding , quadtree or octree encoding) also take a long time to compute by this method.

6-3.2 Method 2 :

The efficiency for the first method can be further improved. The previous method is slow because each time when we get a new sequence after comparing the two sequences, we have to draw and scan an image. Now we will discuss another algorithm where the homogeneous area identification is computed directly from the runlengths. We will discuss two methods,

1. which uses two sequences of runlengths,
2. which uses only one sequence of runlengths.

6-3.2.1 Using Two Sequences of Runlengths:

As usual, before we discuss the method we give the data structure used to store the runlengths. Our data structure is the same as defined above except for a new entry *flag*. The data structure now has six major items,

1. runlength, 2. Sum, 3. Col, 4. flag, 5. left pointer, and 6. right pointer.

we define our data structure as,

structure integer.list(**int** runlength, Sum ;**string** Col, flag ;**pntr** left, right)

The new item *flag* is used in finding the black cell which belongs to the blob in which we are interested. This item i.e.,*flag* may consider three different values,

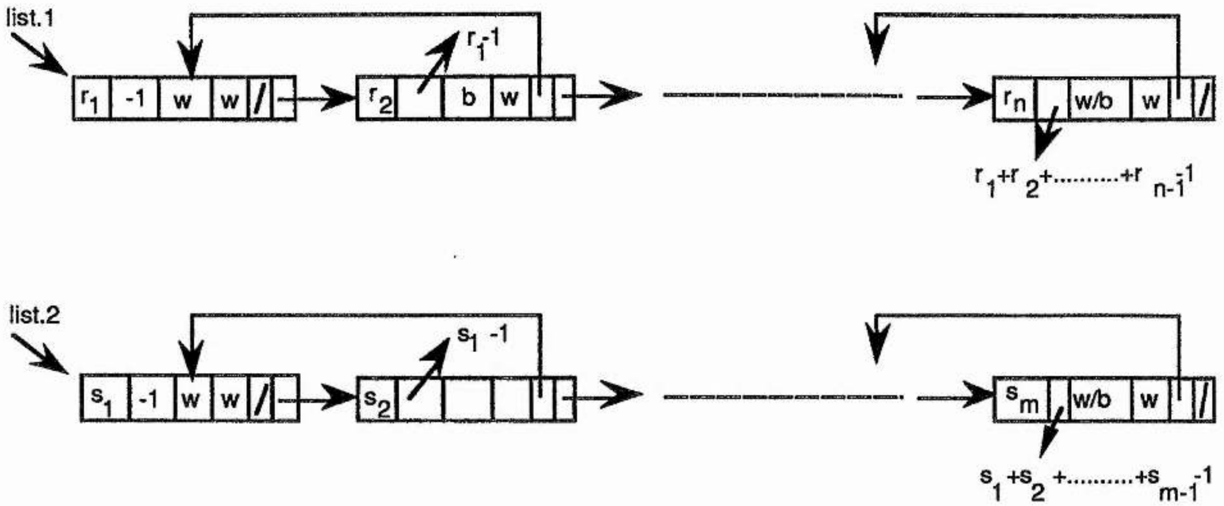
1. Initially we will set it's value to 'w' assuming that there is only one big connected blob.

2. We will set it's value to 'F' if it is a part of the blob in which we are interested.

3. We will set it's value to 'b' if we have used the cell whose flag value was 'F' to find the connected points to it. Once *flag* value is 'b' we will not use that cell again to find the connected points .

Consider the same image in Figure 6.1, and let the area in which we are interested be the same as that considered in the previous method. Let *list.1* be the linked list of n list(or cell) containing $r_1, r_2, \ldots, r_n$ the sequence of runlengths obtained after scanning the image in the horizontal direction and let *list.2* be the linked list of m list (or cells) containing

$s_1, s_2, \ldots, s_m$ the sequence of runlengths obtained after scanning the image in the vertical direction. The data structure used for both lists is the same. Initially we assume that all the cells have the *flag* value equal to 'w'. In other words we are assuming that all the cells initially belong to the blob in which we are interested. *list.1* and *list.2* are shown below,



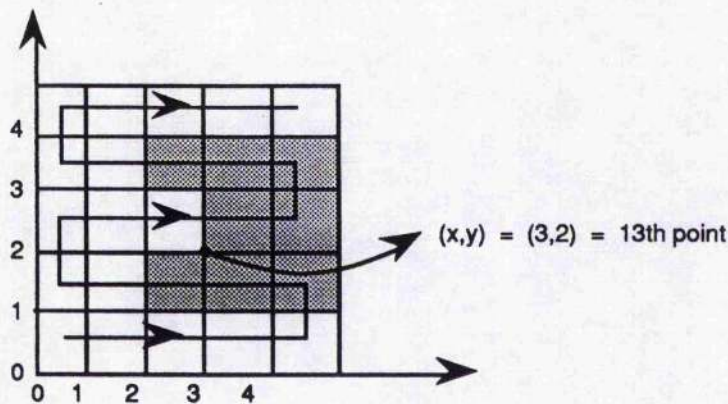
Note : In the first cell we are assuming 'Sum' equal to -1, because the curve starts from the 0th point (see chapter 2).

Now we have two linked lists for the same image, scanned in the horizontal and the vertical directions. The starting point should be a subset of the set containing all the points belonging to the blob in which we are interested. In the last method we used a long runlength to find a starting point. Here we will use the cursor to find the starting point. In PS-algol the standard function *'locator'* returns a structure containing the information about the status of the mouse. With the help of this function we can get the (x,y) co-ordinates for a point belonging to the area in which we are interested. Since a murray polygon is an explicit function. i.e.,

n -----> f(x,y) i.e., if a point on the curve is given then the corresponding co-ordinates of that point can be obtained.

f(x,y) -----> n i.e., if the co-ordinates of a point is given then the corresponding nth point on the curve can be obtained.

we can easily find which nth point it is on the horizontal or vertical murray scans. Using the item *'Sum'* defined in the *structure* we can easily find the cell in which this nth point lies. Since this point belongs to the blob, by the equivalence relation all the points in that cell are connected to each other. Our next step is to change the *flag* value which is 'w' to 'F'. Before we move ahead let us consider an example to discuss the theory explained so far. Consider an image of size 5*5 with a connected component as shown in Figure 6.3. The linked list which is obtained after scanning the image in the horizontal direction is also given in Figure 6.3.



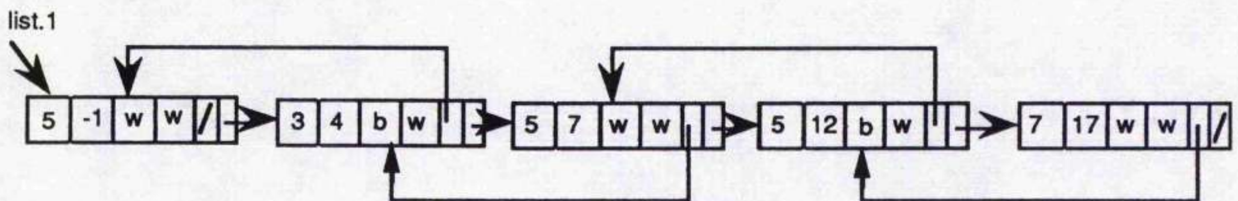
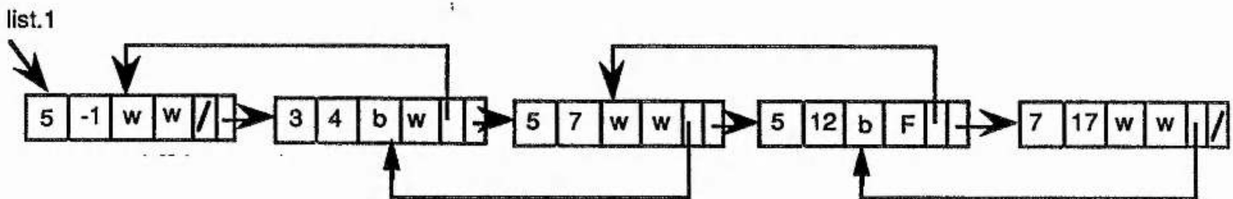
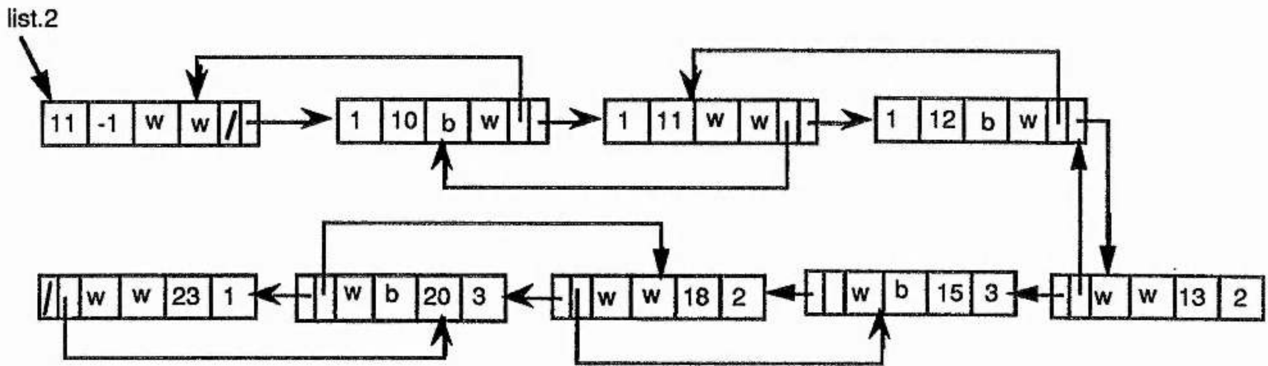


Figure 6.3. A connected image with the runlengths.

Suppose the co-ordinates obtained by using the standard function *locator* are (x,y) = (3,2). Using the transformation explained in chapter 2, we can find the nth point on the curve corresponding to the co-ordinates (3,2). From the image we can easily see that it is the 13th point (*Note : Actually (3,2) is the 14th point on the curve but since the curve starts from 0th point i.e pixel at (0,0) , it is the 13th point on the curve)* . If we see the linked list, this point belongs to the fourth cell. This information is obtained from the item *'Sum'* which tells us that 12 points have been used before that and the points from the 13th to the 17th belong to this cell. In the new list we will change the *flag* value to 'F' indicating that it belongs to the blob, see below,



Now we will scan this new changed list and wherever we find a cell with the *flag* value 'F', we will use those cell points to find the corresponding points in the second linked list (i.e., vertical). Once we have used the points of a particular cell whose *flag* value is 'F' we will then change this value to 'b', indicating that this cell belongs to the blob and has been used for finding the other connected points to it. Therefore in the above list the *flag* value for the 4th cell which is 'F' will turn to 'b' after use. Consider the same image given in Figure 6.3. The vertical linked list i.e., list.2, obtained by scanning an image (Figure 6.3) in the vertical direction is given below.



From the new linked list i.e., list.1, given above we can say that from 13th to 17th points belongs to the blob in which we are interested. But these points will appear in a different numeric order in the vertical murray scan. Using these points the corresponding points on the linked list i.e., list.2, can be obtained.

n ------> f(x,y)

f(x,y) ------> m, where n and m are the points on the two scans.

We find that the 13th point is the 17th point in the vertical scan.This 17th point belongs to the 6th cell of the list.2, this implies all the points in that cell are connected to the points in the corresponding cell of list.1. The next step is to change the *flag* value to 'F'. Similarly consider the 14th, 15th .... points to find the corresponding cells in the second list i.e., list.2. Once we have completely scanned list.1, we will scan list.2 and if we find a cell with *flag* value 'F' we will use that cell to find the corresponding point on list.1. The *flag* value will change to 'b' once a cell has been used. We will keep on doing that until both the lists do not have any cell with the *flag* value equal to 'F'. In the end we will merge those black cells whose *flag* value is 'w' with the

two neighbourhood white cells. The cells whose *flag* value is 'b' are the required runlengths.

Extension to 3-D and n-D Images :

The method is similar to the one explained above. Here instead of two sequences of runlengths, we will consider three sequences of runlengths. The first one corresponds to the front view, the second one corresponds to the left view and the last one corresponds to the bottom view of a 3-dimensional image. By the front view we mean that the planes are parallel to the XY-plane , by the left view we mean that the planes are parallel to the YZ-plane, and for the bottom view the planes are parallel to the XZ-plane. To get a murray scan of that type depends upon the values of the radices. In chapter 2, we have discussed front scanning, which can be done either plane by plane or tile by tile depending upon the values of the radices. If the planes are parallel to the YZ-plane or to the XZ-plane a murray scan can easily be obtained by choosing the appropriate values for the radices.

Let $r_1, r_2, r_3, r_4, r_5, r_6$ be the radices where

r_1 and r_4 belong to the x-radices,

r_2 and r_5 belong to the y-radices,

r_3 and r_6 belong to the z-radices,

If r_1 takes the value 1 then the scan will be forced in the YZ-direction. Similarly if r_2 takes the value 1 then the scan will be forced in the XZ -direction. The image can be scanned in plane by plane fashion or in tile by tile fashion.

Alternatively, if we interchange the position of the x-part and the z-part such that radices r_1 and r_4 now belongs to the z-part, and radices r_3 and r_6 belongs to the x-part then the scan will be forced in the YZ-direction (*Note : since we are interchanging the radices we also have to interchange the digits*). If we want to scan the image in plane by plane order, then the radix

r_3 will take value 1 and if we want to scan the image in tile by tile fashion then we will divide the x-dimensions which can be even, into the suitable factors e.g., if the x-dimension is 9 then the two factors corresponding to r_3 and r_6 can be 3 and 3, for the x-dimension to be 10 the two factors can be 5 and 2 (*Note : radix r_3 can not be even)*. Similarly If we interchange the position of the y-part and the z-part such that radices r_2 and r_5 now belongs to the z-part, and radices r_3 and r_6 belongs to the y-part then the scan will be forced in the XZ-direction (*Note : since we are interchanging the radices we also have to interchange the digits)*. Now to scan the image in plane by plane fashion or in tile by tile fashion the radix r_3 will assume different values as defined above.

Once we have three linked lists obtained after scanning a 3D-image from the three different directions, the problem of connectivity can be easily solved. The approach is similar to the one discussed above. We will consider each scan one after the other and will use those cells which have *flag* value equal to 'F' and turn it to 'b' once used. When all the three lists do not have any cell having a *flag* value equal to 'F', the program will stop. Then after merging those black cell where the *flag* value is 'w', we will get the runlengths for the connected blob. Using these three sequences of runlengths we can obtain the runlengths for the three remaining sides i.e., back side, left side and top side.

6-3.2.1.1 Comparison Between Method 1(using images) And Method 2 (using runlength sequences) :

An Image	Number of Black Pixels	Number of White Pixels	Method 1 (time)	Method 2 (time)
Image a	2875	6926	5.40 secs	2.20 secs
Image b	3134	6667	5.48 secs	3.15 secs
Image c	6246	3555	8.10 secs	8.00 secs

Table 6.1. Comparison betweeen the two methods, where method 1 uses images and method 2 uses two sequences of runlengths.

Different images which are given in chapter 3 are considered to compare the two above methods. The first method takes an image as an input, whereas method 2 takes two sequences of runlengths to find the required connected component. From the result shown in Table 6.1, method 2 is found to be faster than method 1. The reason for that is, in the case of first method we have to draw and scan the image whenever we will get a new sequence of runlengths, which is obtained after comparing the two sequences of runlengths (refer section 6-2.1). In the second case we do not have to consider the images, since the connected component will be obtained straight from the two sequences of runlengths, which are the input values. Both the algorithm are coded in PS-algol.

6-3.2.2 Using One Sequence of Runlengths :

The above two methods discussed so far consider two sequences of runlengths obtained from two different murray scans to find a connected

component. In this section we will discuss a new algorithm which will take only one sequence of runlengths to find the large homogeneous area.

This result was obtained by using a linear murray scan. That is, for an image of size n*m a murray scan which is used with with the radices $r_2\ r_1$ given by n m or m n (*Note : in the second case we have interchange the radices and the digits also , as discussed in chapter 4)*. This forces the scan to move across or up the full width of the image before a unit change in the y-direction or in the x-direction occurs (see chapter 2, and 4), resulting in a scan pattern as given below in Figure 6.4.

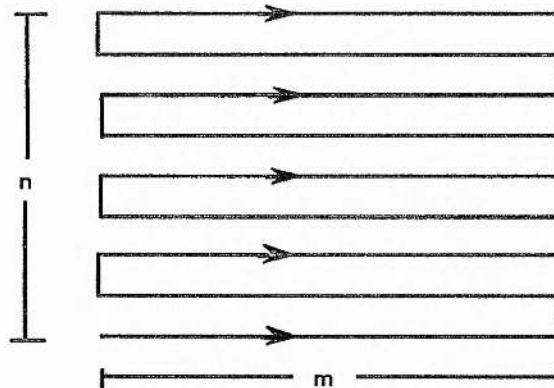


Figure 6.4. Linear scan simulation by murray scan with radices $r_2\ r_1$ given by n m.

This does not have fly-back and the runs are allowed to wrap round one scanline to the next thus giving better results than a system with maximum runlength limited to a scanline length.

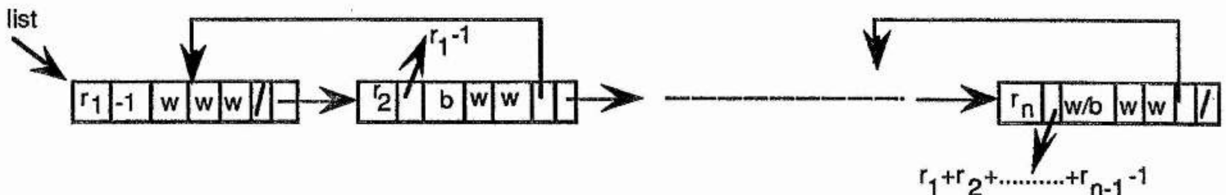
The data structure which has been used has seven items and is defined as,

structure integer.list(**int** runlength, Sum;**string** Color,Lflag,Rflag;**pntr** left,right)

All the item are defined earlier except the *Lflag* and *Rflag* which are used to find the left and the right connectivity. Both the items *Lflag* and *Rflag* will

consider three different values, as discussed before for the item *flag* in the above method.

Let $r_1, r_2, r_3, \ldots\ldots, r_n$ be the sequence of runlengths obtained by scanning the image by using a linear horizontal murray scan. Initially both the flags i.e., *Lflag* and *Rflag* are given value 'w' as we did in the previous method. The linked list is given below,



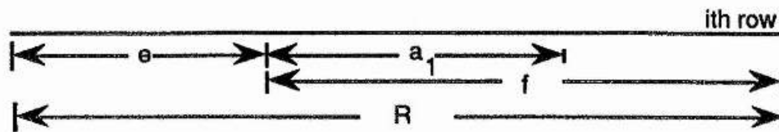
The starting point can be obtained by either of the two techniques discussed above. Once we get the starting point we will find the cell in which it belongs and turn the *Lflag* and *Rflag* values to 'F'. Now we will start from this cell whose *flag* values is 'F'. We will go *left* and *right* in the linked list to find the connected components. Initially we will find the connectivity to the right of the list turning the *Rflag* value to 'b' i.e., it is connected and we do not have to use it again to find the connectivity in the right direction. If we find a cell or cells which is/are connected to this cell we will change its *flag* values to 'F' and will consider the next cell with *flag* value equal to 'F', turning the *Rflag* value to 'b' and so on. Once we hit the end of a list we will go left to find the connected components. This time we will consider those cells whose *Lflag* values are 'F'. When we hit the left end we will go to the right of the linked list. When there is no cell with *flag* values equal to 'F' the process will stop. Finally to get the runlengths for the large homogeneous area we will

merge all the black cells with the two white neighbours if their *flag* values is 'w'.

Implementation :

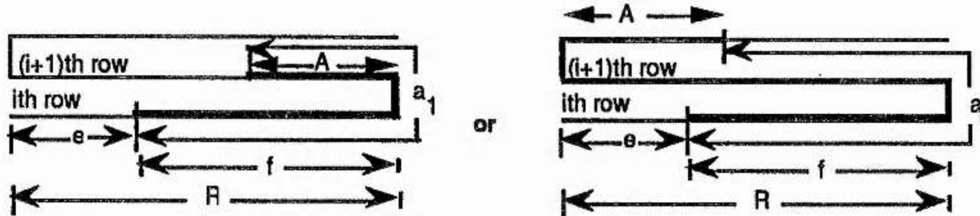
Here two procedures which are used in finding the connected components are discussed. The first procedure i.e., *pixel.before.b.cell,* returns the number of pixels before a black cell in a row and the second procedure i.e., *split.b.cell,* splits the black cell if the runlength obtained for this cell belongs to two or more number of rows. The information obtained from both the procedures is then used to find the parameters for the next black cells to lie in, if they are connected. A simple example to explain this method is given at the end of this chapter.

Let a_1 be the number of white pixels behind the black cell under consideration. Let the start point of a cell with a_1 pixels lies in the ith row. To find the number of pixels behind the cell with a_1 pixels, we have to find the start point for this cell. The start point for a cell can be obtained by using the information stored in the item *Sum* of a list. As discussed earlier, the item *Sum* records the number of pixels used before a cell, hence the starting point of a cell will be given as *Sum+1.* The row number in which this start point lies can be obtained by dividing the start point of a cell by the length of an image. The divisor term will be the required row number (i.e., *row.number = (Sum+1) div R , where R is the length of an image).* The start point for a row will now be equal to *row.number*R.* Now the number of pixels(say *e*) behind the cell in the ith row is equal to the start point for the cell minus the start point of the row. Let *f = R-e* be the remaining pixels in the row.



We now have the number of pixels behind a_1 pixels. Our next step is to find whether the runlength a_1 corresponds to the ith row only or to some other rows also. This can simply be found by comparing the values for a_1 and f.

> If $a_1 < f$ then the runlength a_1 belong to the ith row only(see above example, here ith row has been marked with these values), otherwise it corresponds to some other rows also.
>
> If $a_1 > f$ then a_1 will assume value equal to $(a_1 - f)$ *rem* R $(= A$ *, say)* and *e* will have the value zero. For example,



The procedure *'pixels before.b.cell'* is given below.

```
! Input is 'Sum'  i.e.; number of pixels used,
! the length of the image (R) and the no. of W.pixels (a1)
!The output is the vector of integers.
let pixel.before.b.cell = proc(int Sum,R,a1 -> *int)
begin
   let lst.pnt:=Sum+1
   let find.row:=lst.pnt div R
      let lst.pnt.of.row:= R*find.row
   let e:=lst.pnt-lst.pnt.of.row
   let f:=R-e
```

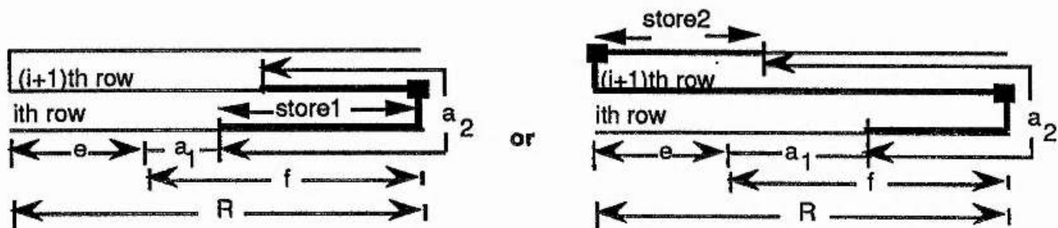
```
if  a1>=f   then @1 of int[0, (a1-f) rem R ]
            else  @1 of int[e, a1]

end
```

Another procedure is used to split the black runlengths if necessary. This is because if a black runlength corresponds to two or more consecutive rows and we are finding connectivity in the right direction i.e., top-direction, then we do not have to consider those rows which are underneath the black rows. Only the one which is on the top should be considered. Exceptions can be arises when the top row partially covers the bottom row. Here we will split the black runlength into two parts and will separately find the connectivity for the two parts. Since these two parts are connected to each other, hence the other parts which are connected to these two parts will be connected to each other. The procedure *split.b.cell* and the different case for a black runlength are discussed below.

Let a_2 be the number of black pixels ahead of a_1. For a_2 two cases are possible,

1. $a_2 >= f-a_1$. We have to divide a_2 into two parts. The first part (say store1) at the ith row will find connectivity with the pixels in the (i+1)th row and the second part (say store2) at (i+1)th row will find connectivity with the pixels in the (i+2)th row.



store1 = R - e - a_1
store2 = a_2 - store1
■ Here a_1 and e is equal to zero

Further, if store2 > store1 -1 but less than R, this implies we can not use store1 to find the connectivity since all the pixels of store1 are covered with the pixels of store2. Hence store1 will be equal to zero. Also store2 starts from one end of a row indicating that there are no pixels behind it. The values for a_1 and e will turn to zero. If store2 > R then we will again adjust store1 and store2 by making store1 equal to R and store2 equal to, (store2 rem R).

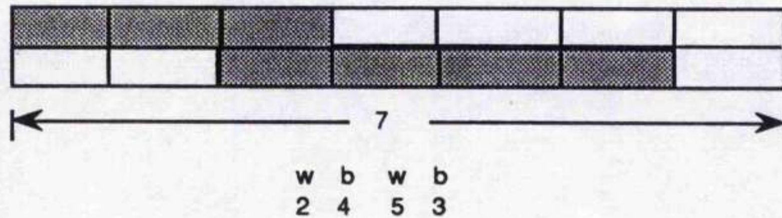
2. If $a_2 < f - a_1$ then store1 will be equal to a_2 and store2 will be zero.

The procedure '*split.b.cell*' is given below,

```
! Input is no. of b.pixels (a2), B = f-a1,
! the length of the image, the term A has been used for the value a1.
! The output is the vector of integers.
let split.b.cell = proc(int a2,B,A,R,e -> *int)
begin
   let store1:=0;let store2:=0
   if a2 >= B then
   begin
       store1:=B;store2:=a2-B
       case true of
       store2>store1-1 and store2<=R : {store1:=0;A:=0;e:=0}
       store2>R : {store1:=R;A:=0;e:=0}
```

```
        default : {}
    end
    else{store1:=a2;store2:=0}
    @1 of int[store1,store2,A,e]
end
```

Now we have the information about the number of pixels behind a black cell in a row, the parameters at which other black cells will be connected may easily be calculated. Consider a simple example, an image of size 7*2 and the corresponding runlengths given below,



w	b	w	b
2	4	5	3

$a_1 = 2$,

the start point for this white runlength = 0 and --- (1)

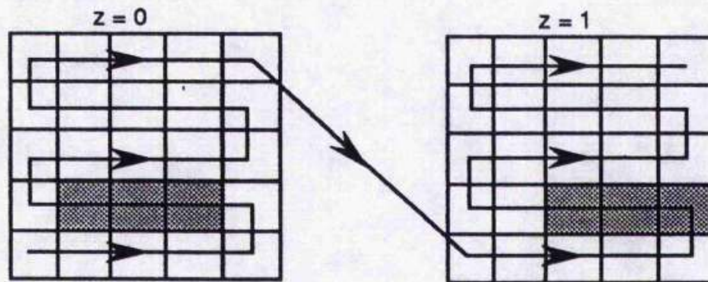
the start point for the row = 0, ------- (2)

the value e = (1) -(2) = 0,

the quantity f = R-e = 7 - 0 = 7. Now since $a_1 < f$ there will be no change for the value of a_1.

a_2 (= 4) and is less than $R-a_1-e$, so store1 = 4 and store2 = 0. Then if the next black runlengths lies between R+F to $R+F+a_2+1$, where $F = R-a_1-a_2-e$, i.e between 8 to 13, then it is connected otherwise not. If we see, the sum of the first three runlengths is 11, that is the next runlength of size 3 starts from the point 12, and since it is black this implies that it is connected.

The same algorithm can also be extended to 3D-images. The only information which we need to include in the previous algorithm is the connectivity with the back objects. In the 2-dimensional case we consider a black runlength and then find the corresponding connected points in the up and down directions. Since in the 3D case the planes are placed one after the other, we need to find the connectivity in the back direction also. Similar argument for the 2D case can be extended for 3D-images. For example, consider an image of size 5*5*2 as shown ,



The corresponding runlength obtained after scanning an image by using a 3D-murray scan, which will scan the image in plane by plane fashion, is given below,

number	--->	1	2	3	4	5
colour	--->	w	b	w	b	w
r.length	--->	6	3	21	3	17

As we can see from the image given above that the black runlength numbered 2 and 4 are connected to each other. Now for the connectivity only we have to find the range for the next runlength to lie in. The range can easily be calculated as discussed above. In the above case, if we select the 2nd runlength as the start point then for the back connection the range for the next runlength will be between 22 and 26. Since the next black runlength i.e., 4th, lie in that range hence it's connected.

6.3.2.3 Comparison Between Method 2(Part 1), And Method 2(Part 2) :

An Image	Number of Black Pixels	Number of White Pixels	Method 1 (time)	Method 2 (time)
Image a	2875	6926	2.20 secs	0.40 secs
Image a	2875	6926	1.20 secs	0.39 secs
Image b	3134	6667	3.15 secs	0.44 secs
Image b	3134	6667	1.05 secs	0.43 secs
Image c	6246	3555	8.00secs	0.52 secs
Image c	6246	3555	1.10secs	0.43secs

Table 6.2. Comparison betweeen the two methods, where method 1 uses two sequences of runlengths and method 2 uses one sequence of runlengths.

The processing time shown above for the two methods has obtained for different connected components in an image. The time difference between the two methods is very large specially in the case of Image c. The reason is discussed below.

6.4 Remarks :

Three different methods which finds the connected componentsin an image have been discussed above. From the results obtained by these three methods it has found that the method with linear murray scan is faster than the rest two methods(see Table 6.1, and Table 6.2). The method 2(part 1) which takes two sequences of runlengths, is slow because some of the pixels are repeated to find the connected component. For example, let the two sets of runlengths, obtained by scanning the image in horizontal and in the vertical direction are,

$r_1, r_2, \ldots\ldots\ldots\ldots\ldots\ldots\ldots, r_n$ ----------------(1)

$s_1, s_2, \ldots\ldots\ldots\ldots\ldots\ldots\ldots, s_m$ ----------------(2)

Suppose the starting point lies in the cell r_i. Using r_i pixels we will find the corresponding cell in the sequence (2) and then these cells in (2) which contain previous r_i pixels will be used to find the corresponding cells in (1) . Here we are repeating r_i pixels which are already used in finding out the connected points. This method can be made faster if we represent each point separately in a sequence. For example, if a point say A lies in the cell s_j then we can break s_j into two or three runlengths. If pixel A is the last or the starting point in the cell s_j then we can write s_j as $(1 , s_j -1)$ or $(s_j -1 , 1)$, otherwise $(s_{j-1} , 1, s_{j+1})$ where $s_j = s_{j-1}+1+s_{j+1}$. But it is not true that the processing speed improves since the size of the sequence increases and so the time to scan the whole sequence. Therefore in speed the method 2(part 2) will always be faster than the other two. But since a point in a general murray scan has four directions to move hence the chances of capturing more pixels of the same colour is more than that of a point in a linear murray scan, which goes from left to right with no fly back. The result will be better compaction in the case of a general murray scan rather than that of the linear murray scan. The method 1 and method 2(part 1) which are slow may be more compact in comparison with the method 2(part 2).

The same approach as given in method 2 (part 2) can be obtained by using the linear scan method. Since a linear scan goes from left to right, a fly back will often result in a break in the runlength and hence will require more space to store a 3D-image. For example, for an image of size 100*100*3 we need 300 linear lines i.e., minimum number of runlengths, to store the whole image. Secondly we have to keep the record for all the scan lines in order to find the connected component. On the other hand a linear murray scan has no

fly back, which can be slightly advantageous in getting the more compact runlengths and secondly the whole image can be represented as a single sequence of runlengths which can be used for further processing . The time of completion should be same with both the methods since only addition and subtraction calculations are required.

In comparison to quadtrees or octrees approaches the time requirement for the method 2(part 2) may approximately be same for finding the connected component. It depend very much upon the shape of the component(i.e.,image) also. Unnikrishnan, and Venkatesh(1984) has shown that for a 64*64 image, where 662 pixels are black, a LHQT takes 1067ms to find the connected areas. Their algorithm is coded in Pascal and has run on DEC 1090. In the case of a linear murray scan it has found that for an image of size 100*100 , where 2875 pixels are black it takes 0.1 sec. Our algorithm is coded in PS-algol. However the scanning part may be time consuming in the case of quadtree or octrees. In case of linear quadtree[Refer Gargantini(1982)], to get the required codes for an image we have to apply condensation and sorting to the collection of codes for the black pixels, which we do not require in the case of a linear murray scan. In comparison to linear murray scan coding, the quadtrees or octrees coding may be more compact, especially linear quadtree or Octree encoding where only black pixels are to be stored. Better compaction may be obtained if we redraw the image after finding the connected area by method 2 (part 2) and then rescanning the image using a suitable general murray scan. The result may be comparable to that of quadtree or octree approaches and the total time may approximately be same for both the methods, since not much time is wasted in scanning the image using the murray approach

Chapter 7

7. **HIDDEN SURFACE REMOVAL AND SHADING** 182

7 - 1 Introduction 182

7 - 2 Hidden-Surface Removal 183

Object-Space Algorithms 184

Image-Space Algorithms 186

List-Priority Algorithms 186

Scan Line Algorithms 195

Scan Line Coherence Algorithms 196

A Visible Surface Ray Tracing Algorithm 197

Octree Methods 198

7 - 3 Hidden-Surface Removal Using Murray Polygons 199

Method 1 201

Method 2 202

Comparison Of Hidden Surface Methods 209

7-4 Shading 210

Introduction 210

Surface Shading Methods 212

Transparency 216

Texture Mapping 219

Antialiasing 219

Shadows 220

7 - 5 Shading Using Murray Polygons 221

Determining The Surface Normal 222

Determining The Intensity Using Murray Polygons 223

Determination Of The Angle Between N And L 225

Smoothing Of Data 232

Results 235

Conclusion 239

7-6 Specular Reflection 240

7 - 5 Remarks 242

7-1 Introduction :

If we consider a 3D-image, we will find that some of the opaque objects and surfaces that are closer to the eye hide other objects from view. The objects which are blocked must be removed in order to render a realistic screen image in real time. The identification and removal of these surfaces is called the hidden-surface problem. The solution for that is to determine the depth and visibility for all the surfaces in an image. Once the surface which is hidden has been removed then the visible surface can be shaded from a given light source. Shaded pictures are produced by recording the shade of gray or the colour of each point in a two dimensional array. Since many shades of gray or shades of colour may appear in an image, corresponding to a visible surface, it is right to call them shaded images.

In this chapter, initially past and present work is categorised and briefly discussed. Two different algorithms, which use murray techniques are discussed to remove the surfaces which are hidden. The algorithms which are generated accept any arbitrary image, (for example, as a CT-scanner provides 3D-data by taking images of a large number of slices through a patient). All the images are black and white and the view plane is considered to be the XY-plane. Since all the planes are parallel to the XY-plane then the x-coordinates and the y-coordinates for the pixels in each plane will be the same but with varying z-value. Shading algorithms are discussed which use diffuse reflection. The results are compared with those obtained by using specular reflection. The algorithms are coded in PS-algol.

7-2 Hidden-Surface Removal :

The task of deciding which parts of an object should be shown and which parts should be omitted was originally known as the "Hidden-Line Problem". Here we eliminate or dash all the lines in an output drawing which were hidden by other objects. Now that shaded pictures are being produced by computer, a different problem which is referred to as, the "Hidden-Surface Problem" has become important. In hidden surface problems one must include or omit entire surface areas rather than just the lines representing edges. Hidden-surface algorithms can be divided into three classes based on the coordinate system or space in which they are implemented[Sutherland, Sproull, and Shumacher(1974)]:

i. Those that compute a solution to the hidden-surface problem in "object-space".

ii. Those that perform calculations in "image-space".

iii. Those that work partly in each, the "list-priority" algorithms.

Object-space algorithms are implemented in the physical coordinate system in which the objects are described. Very precise results, generally to the precision of the machine, are available. These results can be satisfactorily enlarged many times. They are particularly useful in precise engineering applications. Image-space algorithms are implemented in the screen coordinate system in which the object are viewed. Calculations are performed only to the precision of the screen. As shown by Roger(1985), the computational work for an object-space algorithm that compares every object in a scene with every other object in the scene is equal to the number of objects squared (n^2), and for an image-space algorithm which compares every object in the scene with every pixel location in screen coordinates is equal to

nN, where, n is the number of objects (i.e., volumes, planes, edges) in the scene, and N is the number of pixels. For n<N, object space algorithms will require less work than image space algorithms. Since the resolution of the screen is fixed and not very large (e.g., the resolution of Sun 3/60 is 1152*900), it would be better to implement all the algorithms in object space. In practice, this is not the case, image space algorithms are more efficient because it is easier to take advantage of coherence in a raster scan implementation of an image space algorithm. Secondly the cost of the object space algorithms grows as a function of the complexity of the environment, but the cost of the image space is limited because the number of screen dots remains constant, independent of the environment complexity. The list priority algorithms operate in both object and image space. In particular, the "list-priority" calculations are carried out in the object space and the result written to an image space frame buffer. The use of a frame buffer is critical to the algorithm, since each element of a scene is written to a frame buffer in turn. Those elements which are closer in a list will overwrite the contents of the frame buffer, thus solving the hidden surface problem.

The following section examines several object and image space algorithms. Each algorithm illustrates one or more fundamental ideas in the implementation of hidden-line/hidden-surface algorithms.

7-2.1 Object-Space Algorithms :

Robert(1963) devised the first known solution to the hidden-line problem. His algorithm tests each relevant edge to see if it is obstructed by the volume occupied by some object that lies between the edge and the viewpoint. The algorithm thus capitalizes on the spatial coherence of objects: *it tests edges against object volumes.* This test is implemented by writing a

parametric equation for a line from a point on the edge to the view point. The equation is as given,

$$P(t,b) = (1-t)A_1 + t A_2 + b v \qquad (1)$$

$$0 <= t <= 1; \ 0 <= b$$

The first two terms represents the parametric equation of a point on the edge A_1A_2 in the perspective coordinate system and the third term (i.e, v), is a vector pointing toward the viewpoint in the perspective space, $(0,0,-\infty)$.

The next step is to find whether the point P(t,b) lies inside a convex object. This can be determined by simply finding, whether the point P(t,b) lies "inside" of all planes that comprise the object or not. If it is inside then it lies inside a convex object, otherwise the object will be broken into a number of convex objects, which is very tedious. The condition for finding this is,

$$P(t,b) \, . \, Eij <= 0 \text{ for all } i \qquad (2)$$

where Eij is the plane equation of the ith face of the object j. If for a given object j, values of t and b can be found that satisfy (2), the point on the edge corresponding to t is hidden by the object. For minimum and maximum values of t various techniques are used to solve equation(2).

This edge/object test may discover that:

i. the edge is entirely hidden by the object.

ii. no portion of the edge is obscured by the object.

iii. one part of the edge is not obscured, or

iv. two portions of the edge are not obscured. Any unobscured portions are then tested against the remaining objects.

The disadvantage of Roberts algorithm is that it restricts the environment, all volumes or objects in a picture should be convex. If an object is not convex then the algorithm will firstly represent it by a collection of convex ones, which is a difficult task [refer Robert(1963)].

7-2.2 Image-Space and List-Priority :

The image-space and list-priority algorithms are designed to create images for a fixed resolution display, often a television monitor. Although the specific aims of the various algorithms are not identical, the group has been motivated by desires for real-time speed and for realism in the images. These algorithms are now used to generate quite spectacular shaded pictures in color.

In *image-space* algorithms, the visibility is decided point by point at each pixel position in the image. The depth of the various surfaces that would be penetrated by a viewing ray at a particular point in the image, is calculated and then the depths are compared for the visibility test. Thus, these algorithms can be capitalize on the lateral separation of the image to reduce the number of depth computations required.

The *list-priority* algorithms, on the other hand, precompute in object-space a visibility ordering or "priority" for all surfaces before generating the picture in image-space. The priority of a surface can be expressed as a linear-ordering of the surfaces such that if ever two surfaces need to be compared for visibility, the one with the lower or higher priority is the visible one. A few examples are illustrate below,

For a simple scene, such as shown in Figure 7.1a, obtaining a definitive depth priority list is straightforward. Here the polygons can be sorted by either their maximum or minimum z-coordinate value. However for the scene

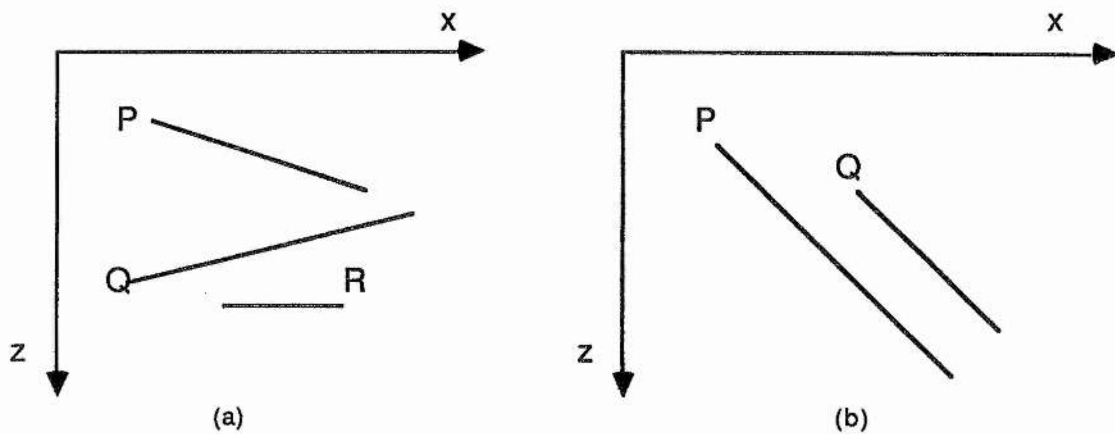


Figure 7.1. Polygonal priority.

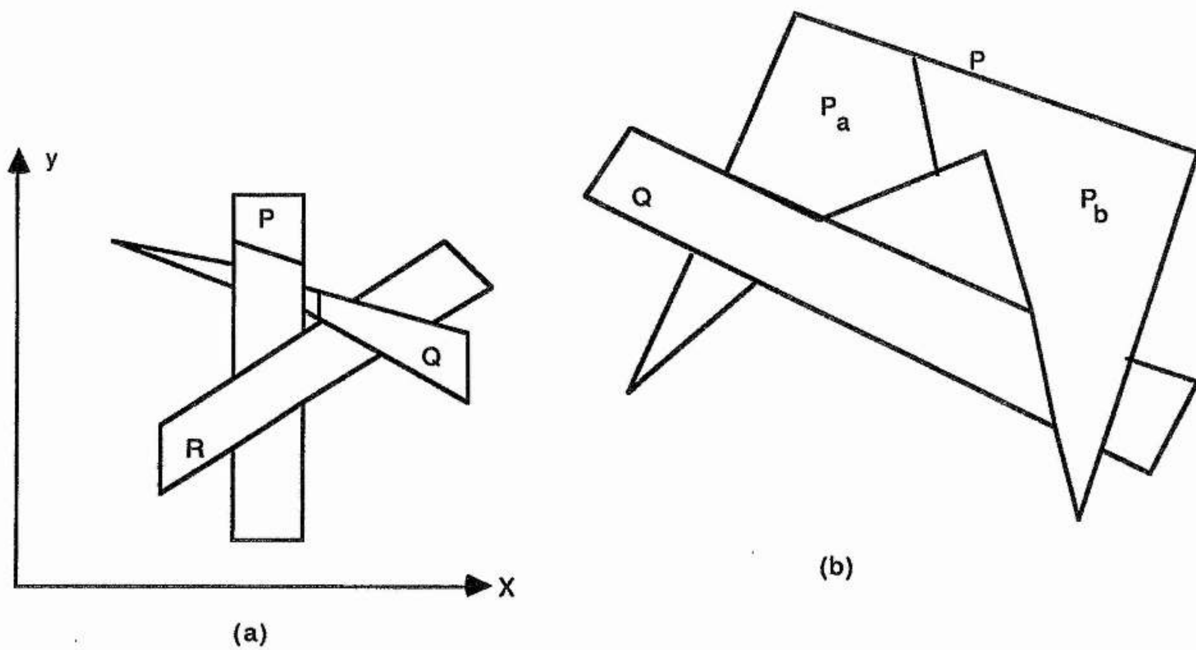


Figure 7.2. Cyclical overlapping polygons.

shown in Figure 7.1b, a depth priority list cannot be obtained by simply sorting in z. If P and Q in Figure 7.1b are sorted by minimum z-coordinate value, then P appears on the depth priority list before Q. The correct order in the priority list is obtained by interchanging P and Q. As illustrated in Figure 7.2, the polygons cyclically overlap each other. In Figure 7.2a, P is in front of Q which is in front of R which in turn is in front of P; similarly in Figure 7.2b. Here a definitive depth priority list cannot be immediately established. The solution is to cyclically split the polygons along their plane of intersection until a definitive priority list is obtained. This is shown by dashed lines in Figure 7.2.

The following sections examine several image-space and list-priority algorithms in detail. Each algorithm illustrates one or more fundamental ideas in the implementation of hidden-line/ hidden-surface algorithms.

7-2.2.1 Image-Space Algorithms :

The Warnock(1969) algorithm assumes that sample areas on the screen, called windows, can be declared to be homogeneous if ;

1) no faces fall within the sample window,

2) one face completely covers the window and is nearer the viewpoint than every other face that falls in the window.

If the window under consideration is not homogeneous, then it is divided into four smaller sample windows, and each of these is examined similarly. When the size of the sample windows decreases to the size of the raster element, the subdivision process is terminated (see Figure 7.3).

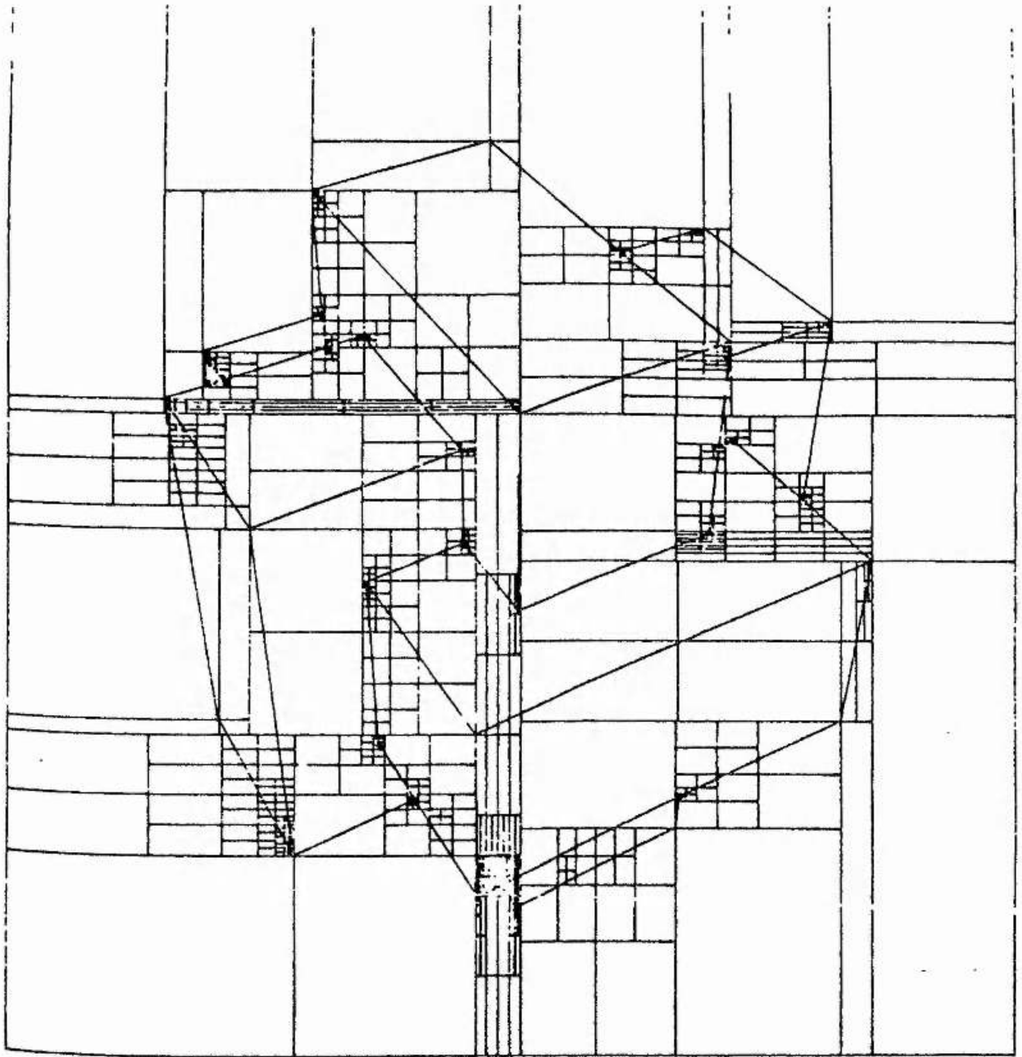


Figure 7.3. Subdivision by Warnock's algorithm. The object contains three intersecting bricks (Sutherland, Sproull, and Schumacker(1974)).

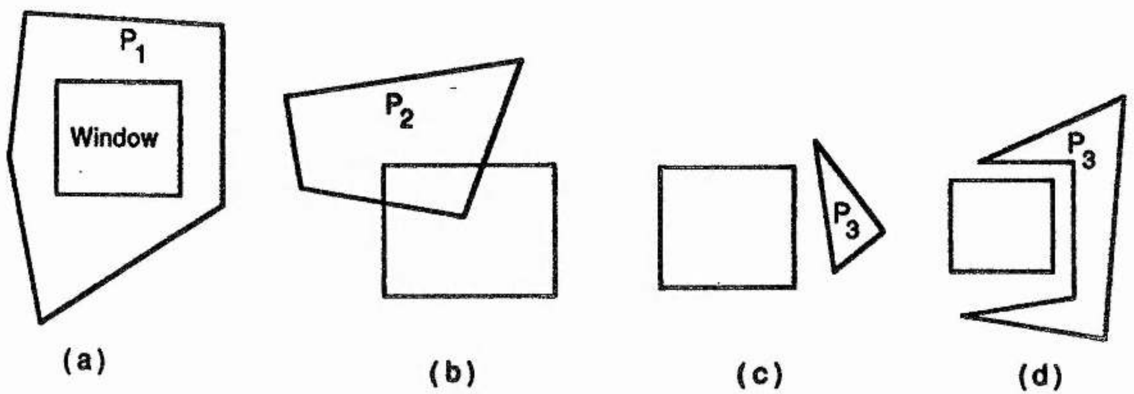


Figure 7.4. The relationship between a face (or polygon) and a sample window.
a. polygon P_1 surrounds the window,
b. polygon P_2 intersects the window,
c, & d. polygon P_3 is disjoint from the window.

A set of faces is compared to the window to see whether the face;

1) surrounds the window,

2) intersects the window, or

3) is completely disjoint from the window (see Figure 7.4).

An important concept of the Warnock algorithm is that the hypothesis test for a sample window need not test all faces in the environment. If a hypothesis test fails, the four sub-windows to be examined need only be tested against intersectors of the original window since faces disjoint from the large window will certainly be disjoint from the four small windows, and faces which surrounded the original window will surround its decendant windows.

The faces are grouped into two categories,

1) those that are disjoint from this window,

2) those that are relevant to this window.

The relevant faces are then passed down to sub-windows, where the faces are again compared with the subwindows. The process terminates when a window is proven to be homogenous.

There are many advantages of the Warnock algorithm; the windows do not need to be rectangular; we can subdivide the windows at specific points, such as vertex locations, rather than at the center point.

One difficulty with the Warnock algorithm is that its output cannot conveniently be passed to a raster-scan device like a television. The decisions about windows are reached in a random order, rather than in a top-to-bottom

left-to-right order. Cohen, Tom, and Rosenfeld(1980) have devised a scheme for driving a raster display from window computations, but it involves a massive sort of the windows by X and Y coordinates.

Warnock's algorithm does not produce the display data in a sequential order as defined above. This defect can be removed if we have a procedure which divides the space into separate regions, such that two consecutive regions are in next door neighbour order. Griffith(1984) used a Hilbert curve, since the smallest window or region emerges naturally from the recursion one after the other, thus adjacent to each other. Griffith, proposed a table driven algorithm for the subdivision purposes. Here a window is divided into two parts. A window should either be divided horizontally or vertically, depending upon the quadrant priority(i.e., the four basic orientations). Four basic orientations of Hilbert polygons are given in Figure 7.5. Types A and D have vertical dividing lines, whereas for types B and C horizontal dividing lines are used (see Figure 7.5). Quadrant priority also determines which half should be dealt with first. The arrows marked in each case indicate the window priority after subdivision.

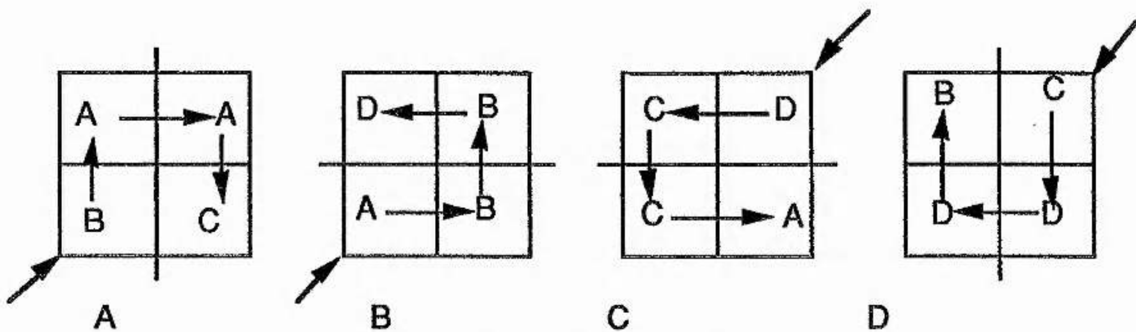


Figure 7.5. The relation between the four basic types of window and their subdivision. (arrows indicate window priority after subdivision).See Griffith(1984).

Weiler and Atherton(1977) tried to minimize the number of subdivisions in a Warnock-style algorithm by subdividing along polygon boundaries. This algorithm is based on the Weiler and Atherton(1977) concave

clipping algorithm. Here the polygon to be clipped is the subject polygon and the clipping region is the clip polygon. The new boundaries created by clipping the subject polygon against the clip polygon, are identical to a portion of the clip polygon. No new edges are created, hence the number of resulting polygon is minimised. It operates in object-space. The hidden-surface algorithm has four steps :

1) A preliminary depth sort.

2) A clip or polygon area sort based on the polygon nearest the eyepoint.

3) Removal of the polygons which are behind that nearest the eyepoint.

4) Recursive subdivision, if required.

The first polygon on the preliminary depth sorted list is used as the clip polygon and the remaining polygons including the clip polygon on the list are subject polygons. Each of the subject polygons is clipped against the clip polygon. Two lists are established: an inside list and an outside list. The portion of each subject polygon inside the clip polygon is placed on the inside list otherwise on the outside list.

Compare the depth of each vertex on the polygons which are in the inside list with the minimum z-coordinate (Zcmin) value for the clip polygon. All the subject polygons on the inside list are said to be hidden by the clip polygon if none of the z-coordinate values of the polygon on the inside list is larger than Zcmin (see Figure 7.6). These polygons are eliminated and the inside polygons list is displayed. Note that here the only remaining polygon on the inside list is the clip polygon. The algorithm continues with the outside list.

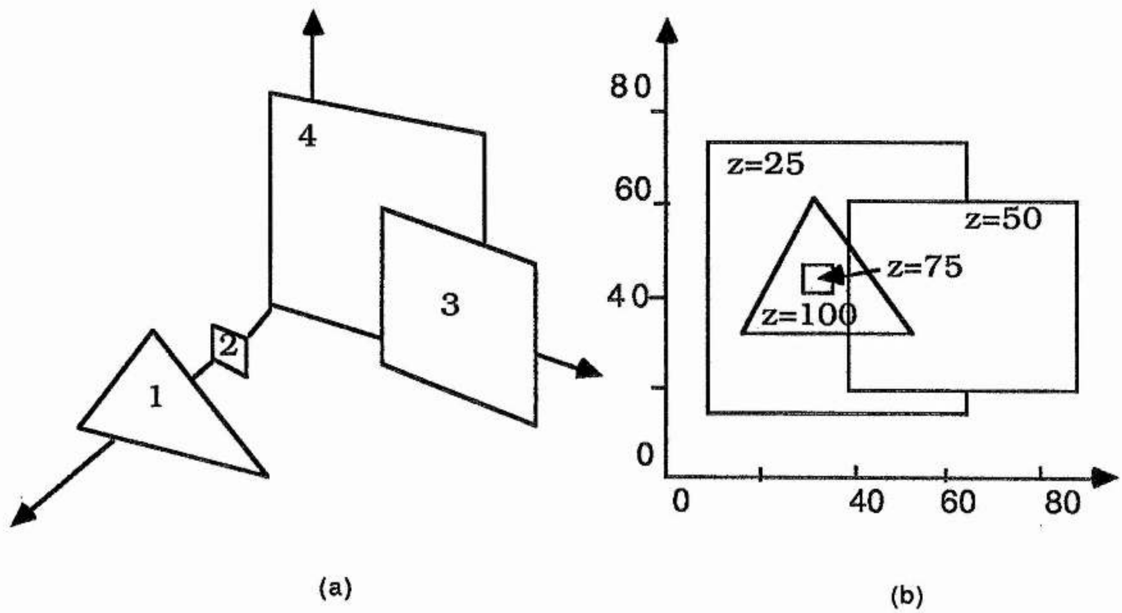


Figure 7.6.
Priority polygon clipping for the Weiler-Atherton hidden surface algorithm.

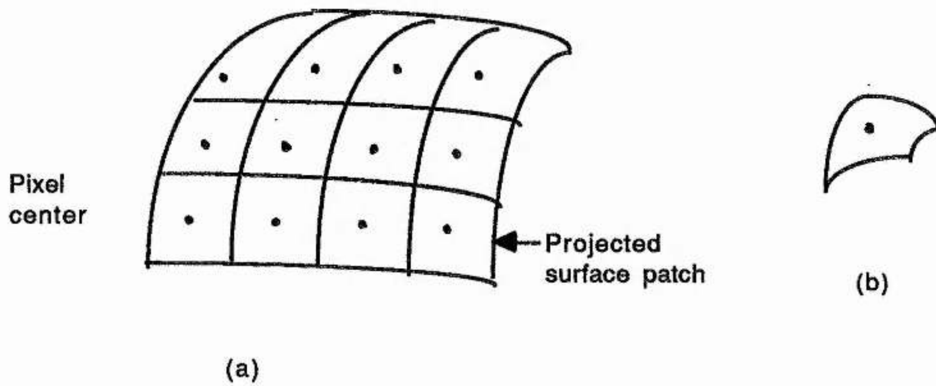


Figure 7.7. Curved surface subdivision.

The subject polygons on the inside list lie at least partially in front of the clip polygon if the z-coordinate for any polygon in the inside polygon is greater than Zcmin. Hence the algorithm recursively subdivides the area, using the offending polygon as the new clip polygon. The inside list is used as the subject polygon list. Since the new clip polygon is a copy of the complete original polygon, it minimizes the number of subdivisions.

All the algorithms thus far described are for objects defined by the planar polygonal faces. Objects defined by curved surfaces must first be approximated by many small facets before any of the algorithms can be used. Catmull(1974a) has developed a Warnock-style subdivision algorithm for curved surface display. Catmull applied the algorithm to bicubic surface patches. Warnock's algorithm recursively divides the image space whereas the Catmull algorithm recursively subdivides the surface (see Figure 7.7). His algorithm is:

1. Recursively subdivide the surface into subpatches until a subpatch, transformed into image space i.e., covers at most one pixel center.

2. Find the intensity of the surface at this pixel and display the pixel.

The efficiency of the algorithm depends on the efficiency of the curved surface subdivision technique. The disadvantage of this method is that it does not present the result in scanline order, which is inconvenient for raster scan line order. Cohen, Lyche, and Riesenfeld (1980) suggest a more general technique for B-spline surfaces, as cited by Rogers(1985).

z-Buffer Algorithm:

The z-buffer approach was proposed by Catmull(1974b). It is implemented in image-space. It is a simple extension of the frame buffer idea. A frame buffer is used to store the attribute or intensity of each pixel in the image space. The z-buffer is a separate depth buffer used to store the z-coordinate or depth of every visible pixel in the image space. The initial value may be thought of as the z position of the background. Polygons will entered one by one into the frame buffer. The depth or z value of a new pixel to be entered into the frame buffer is compared to the depth of that pixel stored in the z-buffer. If the new pixel is in front of the pixel stored in the frame buffer, then the new pixel is written to the frame buffer and the z-buffer updated with the new z value, otherwise no action is taken.

Since image space is of fixed size, the increase in computational work with the complexity of the scene is at most linear. No sorting is required, since elements of a picture can be stored to the frame buffer or z buffer in arbitrary order. Hence the computation time associated with a depth sort is eliminated.

The amount of storage required is the principal disadvantage of the algorithm. It requires a lot of memory (one entry for each pixel), and each entry must have a sufficient number of bits to distinguish the possible z-values. It can also be time consuming since a decision must be made for every pixel instead of for the entire polygon. However it is a very simple method, simple enough to implement in hardware to overcome the speed problem. Further the time required is proportional to the number of objects in the scene. But since the cost of memory is dropping very fast, it makes this method an increasingly popular approach for the hidden-surface problem.

7-2.2.2 List-Priority Algorithm :

The principal contribution of the Newell, Newell and Sancha(1972a) method is the development of a priority computer. As defined above, the priority list is used to determine the face that is visible at any spot. Newell views the list in quite a different way: if we write the images of successively higher priority faces successively onto a picture buffer, the picture buffer will have a correct hidden-surface view after we have processed the entire list. Faces of higher priority will overwrite the faces of lower priority.

The Newell-Newell-Sancha algorithm for polygons is :

The first step in the procedure, sorts all faces by the depth of the farthest vertex of each face. The first face on the list is the one which has the smallest value of the z-coordinate, Zmin The polygon which is farthest from the viewpoint is labelled as P and the next polygon on the list is labelled as Q. If faces do not overlap in depth at all, this sort successively establishes the priority (see Figure 7.1), otherwise we have to test whether the depth sorted list is indeed in priority order (see Figure 7.2) i.e., to examine the relationship of P and Q.

If the nearest vertex of P, Pzmax is farther from the viewpoint then the farthest vertex of Q, Qzmin, then P cannot hide any part of Q. Write P on the frame buffer (see Figure 7.1).

If Qzmin < Pzmax then P obscures not only Q but also any face or polygons on the list for which Qzmin < Pzmax. These faces can be represented as the set {Q}. It may also be possible that P will not hide any part of any polygon in the set {Q}.

If P obscures some polygons in the set {Q}, then P cannot be written in the output buffer. Here interchange P and Q, marking the position of Q on the

list. Repeat the tests with the rearranged list. If Q cannot be written before P and P cannot be written before Q, the priority computer must divide either face P or Q to eliminate the conflict. This conflicts is often called cyclic overlap. In Figure 7.2 face P has been divided by the plane of face Q into two faces Pa and Pb. These two faces are placed in the priority list; the priority computer will then determine the order of Pa, Q, Pb in the correct priority order.

The Newell-Newell-Sancha algorithm for the hidden-surface problem, process all the polygons in the scene, for each frame being presented. If the scene is complex and the frame rate is very high, as in real-time simulation systems, the investment in computing the priority list from the scene is quite high. However, for many real time simulation problems, e.g., flight simulation, where the environment rarely changes, the viewpoint changes quite frequently. Schumacker et al(1969) take advantage of several more general priority characteristics to precompute, off-line, the priority list for simulations of such static environments.

The Schumacker(1969) algorithm allows only convex polygons in the scene. These polygons are grouped into clusters of polygons that are linearly separable. Clusters are said to be linearly separable if a nonintersecting, dividing plane can be passed between them (see Figure 7.8). He refers to the separating planes as a and b. They divide the scene into four regions. The tree structure shown in Figure 7.8b establishes the cluster priority for the scene. Cluster priority can be precomputed. Substituting the coordinates of the viewpoint into the equations of the separating planes locates the appropriate node in the cluster priority tree. The computation of face priority requires computing whether face A can, from any viewpoint, hide face B. If so face A has priority over face B. In the case of cyclic overlap, the cluster will have to

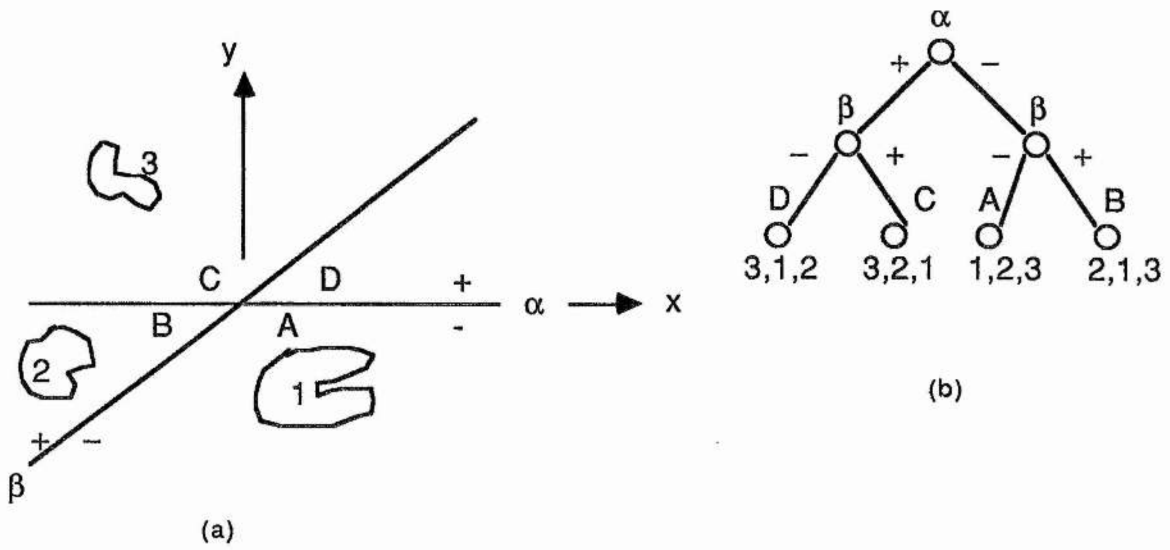


Figure 7.8. Cluster priority.

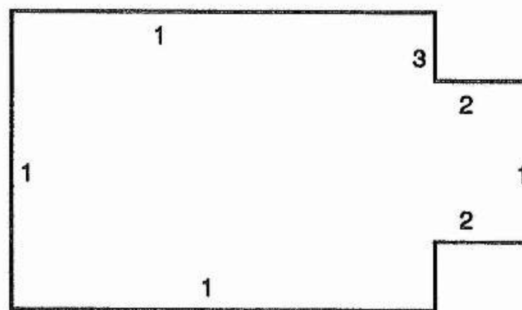


Figure 7.9. Face priority. Top view of an object with face priority numbers assigned(the lowest number corresponds to the higher priority).

be split manually into smaller clusters, then an appropriate hidden surface technique as described by Newell can be applied.

The notion that face priority within a cluster can be computed independent of the viewpoint is one of the major contributions of the Schumacker algorithm. It allows precomputation of the entire priority list. Consider the top view of an object, as shown in Figure 7.9, for which the individual polygonal priority can be precalculated. The priority of each polygon is established by considering whether a given polygon can hide any other polygon from the viewpoint. The more polygons that a given polygon can hide, the higher priority it has. To establish the polygonal priority within a cluster for a given viewpoint, the self-hidden polygons are first removed. The remaining polygons are then in priority order as shown in Figure 7.9.

7-2.3 Scan Line Algorithms :

The Warnock, z-buffer, and list priority algorithms process scene elements or polygons in arbitrary order with respect to the display. The scan line algorithm, as originally developed by Wylie et al(1967), Bouknight(1970) and Watkins(1970), process the scene in scan line order. It operates in image space.

Scan line algorithms take advantage of coherence between successive scan lines and of span coherence within a scan line. They also simplify the geometric calculations by reducing a three-dimensional problem to a two-dimensional comparison of segments in the xz plane. The performance of scan-line algorithms is primarily related to the complexity of the visible image. A scan plane is defined by the viewpoint at infinity on the positive z axis and a scan line, as shown in Figure 7.10). The intersection between the scan plane and the three-dimensional scene defines a one scan line high window. The hidden surface problem is solved in this scan plane window. Figure(7.10b)

shows the intersection of the scan plane with the polygons. The hidden surface problem is reduced to deciding which line segment is visible from each point on the scan line, as illustrated in the Figure 7.10.

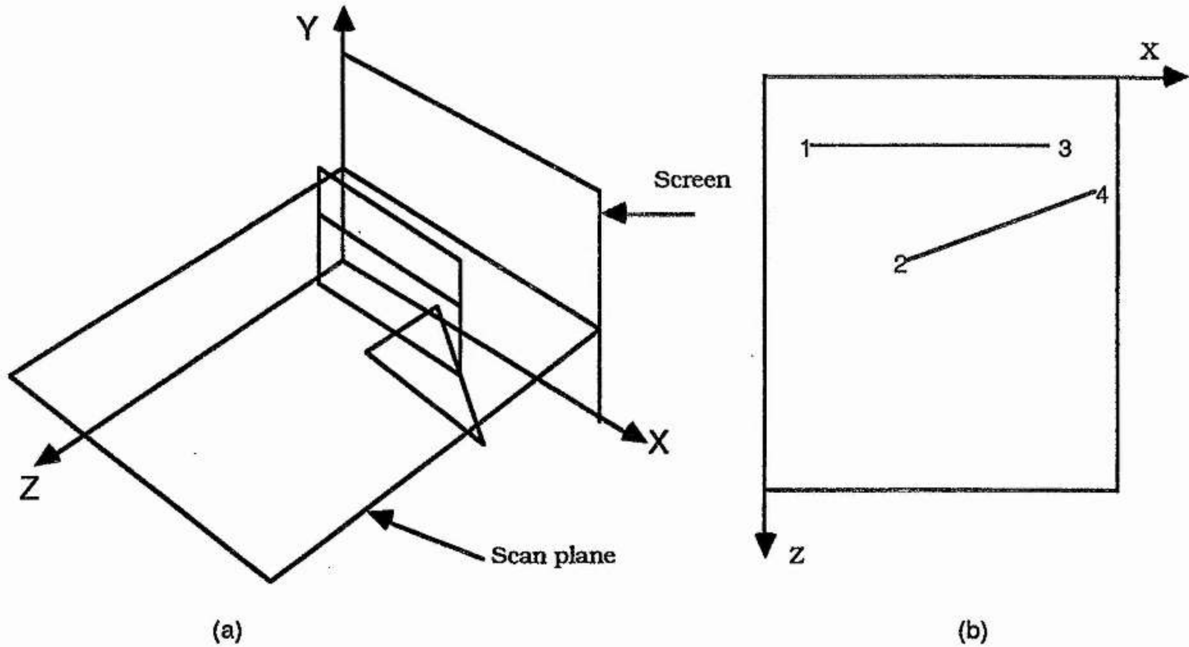


Figure 7.10. Scan plane.

7-2.3.1 Scan Line Coherence Algorithms[Newman, and Sproull(1979)] :

Scan line algorithms solve the hidden surface problem one scan line at a time, usually processing scan lines from top to bottom or bottom to top of the display. The algorithm successively examines a series of windows on the screen. Each window is one scan line high and as wide as the screen. Here two arrays are required intensity[x] and depth[x], to hold values for a single scan line.

For each scan line perform the following steps:

1) For all pixels on a scan line, set depth[x] to 1.0 and intensity[x] to a background value.

2) For each polygon in the scene, find all pixels on the current scan line y that lie within the polygon.

   For each of these x values :

   i. Calculate the depth z of the polygons at (x,y).

   ii. If z < depth[x], set depth[x] to z and intensity[x] to the intensity corresponding to the polygon's shading.

3) After all polygons have been considered, the values contained in the intensity array represent the solution, and can be copied into a frame buffer.

Here a depth value must be computed and compared with the value already recorded in the frame buffer. The algorithm concurrently scan converts all polygons in the scene using one scan line at a time.

7-2.4 A Visible Surface Ray Tracing Algorithm :

In this technique, an object is viewed by means of light from a source. Light rays strike the object and then reaches the observer or viewpoint. The light may reach the observer, by reflection from the surface, or by refraction through the surface. If we trace the light rays from the source, very few rays will reach the viewpoint. Since finding a ray which reaches the viewpoint after striking the object, is totally mathematical, the chances of an error is possible and secondly many sub rays will be generated when a ray hits an

object. Appel(1968) suggested that the rays should start from the apposite end i.e., from the observer to the object.

It works in image space. The viewpoint is assumed to be at infinity, on the z axis. Since the viewpoint is at infinity, all the light rays will be parallel to the z axis. For each ray to be traced we have to find, whether a ray intersect the objects of the scene. If true, find the intersection points for all the objects, which are intersected by a ray. Compare the depth of each of the intersected points, and the point with minimum z value will be the visible point. Display the point using the intersected object's attributes. Finding the intersection of a ray with different objects is very time consuming. In the case of mathematically defined objects, the amount of work can be reduced by using the bounding volume for the object. A bounding volume can either be a bounding box or a bounding sphere, which will cover an object of the scene. This will reduce the number of pixels to be examined.

7-2.5 Octree Methods [Hearn, and Baker(1986)] :

When an octree representation is used for viewing the three dimensional scenes, hidden surface removal is done by projecting octree nodes onto the viewing surface in a front to back order. The front face of a region of space i.e., the side towards the viewer, will have octants 0, 1, 2, and 3. The surface which is in front of these octants is visible to the viewer, whereas octants 4, 5, 6, and 7 may be hidden by the front surface.

Back surfaces are removed by processing data elements in the octree nodes in the order 0,1,2,3,4,5,6,7. Since this results in a depth first traversal of the octree, the nodes representing octants 0,1,2, and 3 for the entire region will be visited before the nodes representing octants 4,5,6 and 7. Similarly, the nodes for the front four suboctants of octant 0 are visited before the nodes for the four back suboctants.

When a color value is found in an octree node, the pixel area in the frame buffer is assigned that color value, provided no value has previously stored at that point. Only those pixels which are visible will be loaded into the frame buffer. If the area is void, we will neglect it. If a node is completely obscured by other nodes then we will neglect that node for further processing. This will help in not wasting time on accessing its subtrees. Further different views of objects can be obtained by applying a transformation to the octree representation. This transformation will reorient the object according to the view selected. It is assumed that all the time the octants 0,1,2, and 3 of a region will form the front face.

The octree method for removing hidden surfaces from the scene is very fast. Only integer additions and subtractions are used and there is no need to perform any sorting or intersection calculations. Another advantage of this technique is that they store the entire solid region of an object. It can be useful for obtaining cross-sectional slices of solids.

7-3 Hidden-Surface Removal Using Murray Polygons :

Here we will discuss two methods, one which scans all the planes using a suitable murray scan and then merges them together, and the second one which scans only the first plane and then merges other planes according to the transparent area (i.e., in the case of black and white, if a pixel is black then there is no need to consider other pixels which are behind of it, but if it is white we will consider the same pixel in the next plane (*Note: The (x,y) co-ordinates for a pixel in each plane will be the same since all the planes are parallel to the XY-plane)*), and so on until a black pixel or the last plane of an image is encountered. In both methods compression of data is done in two stages. Firstly we compress the 3D-image by reducing it into the collection of runlengths. This compression is exact i.e., using the runlengths the same

3D-image can be regenerated and no information is lost. The collection of runlengths are then scanned to give the runlengths for a 2D-image (i.e., we transform a 3D-image to a 2D-image by working on the runlengths obtained by scanning a 3D-image. Once a 3D-image has been reduced to a 2D-image, it can not be rebuilt to a 3D-image.

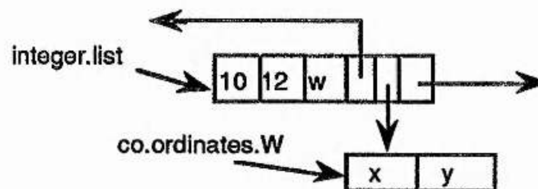
Before we discuss the methods, as usual we would like to discuss the data structure used to store the pixels with different depths. Here we use three different structures. The main structure *'integer.list'*, has six major items namely,

1. runlength 2. Sum 3. Colour 4. left pointer 5. middle pointer 6. next pointer

All the items used in this structure have been defined earlier. The new pointer i.e. middle, points at two different structures and is discussed below,

1. If the colour of a cell of a list is white then the middle pointer will point at the x and the y coordinates of the first pixel of that cell. It has been shown earlier that we can calculate the coordinates if the nth point on the curve is given to us, but since we have to calculate murray integers, gray coded integers, etcetera the process can be very slow. If we know the coordinates of the first pixel of cell the efficiency can be increased. Its use is explained ahead in the method 2. We define this data structure as:

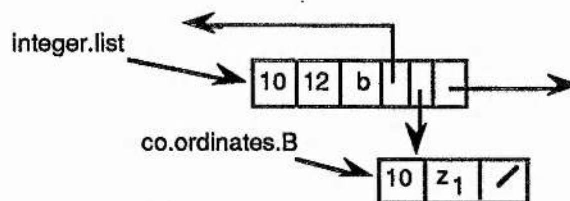
structure co.ordinates.W(**int** x,y) , see below,



2. If the colour of the cell is black then the middle pointer will point at the corresponding depth of the black pixels. The pixels can then be shaded according to the depth stored. We define the data structure as,

structure co.ordinates.B(**int** runs,z;**pntr** right)

For example, suppose there are 10 consecutive black pixels all are from the z_1 plane then the whole structure for storing this will be,



7-3.1 Method 1 :

In this method we scan all the planes one by one and then merge them together to give a single list. We can either use a 3D-murray scan to scan the image in plane by plane order or we can scan each plane separately using a 2D-murray scan. In the second case a murray scan will use the same radices to scan the different planes. Once all the planes are scanned the procedure *'union'* discussed in chapter 5 can be used to merge them together to give runlengths for a single plane. The only point which we have to add in the procedure*'union'* is the information about the z-values. In a list where the cell colour is 'w' i.e white, the middle pointer will get the *nil* value and for a 'b' i.e.,black cell the middle pointer will point at the corresponding depth of the pixels.

The advantage of this method is that we do not need to store the whole image inside the system. The planes can be scanned one by one as discussed above, and the depth can be stored for each plane. Another advantage of this method is that the entire solid region of an object is available for display, which makes this method useful for obtaining cross-sectional slices of solids. Since all the planes are parallel to each other, to get an inside view one can start from the nth plane where,

$$0 < n < \text{total number of planes.}$$

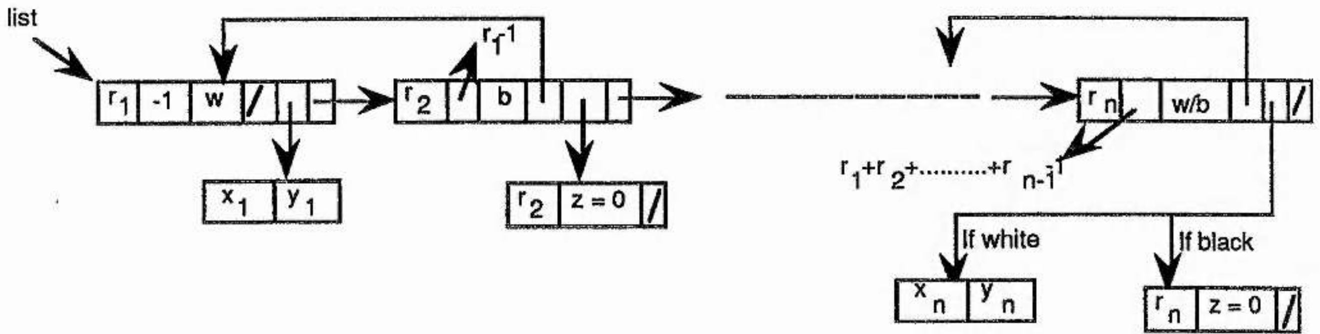
The disadvantage of this method is that we have to consider all the points in a 3D-image. Actually there is no need to consider all the points. For example, if in the first plane a pixel at point (x,y) is black then there is no need to consider the same point in the other planes, since they are hidden by that pixel. The visibility is terminated till all the planes are scanned. Time can be saved by not considering those pixels which are hidden by other pixels. We will use this idea in the next method.

7-3.2 Method 2 :

Here initially we will scan the first plane of a 3D-image. For scanning the image we can either use a horizontal murray scan or a vertical murray scan. The output will be the collection of runlengths with associated colour. The colour information can be removed in the case of black and white images, by assuming that the first runlength will always corresponds to white, as discussed earlier.

Let $r_1, r_2, r_3, \ldots\ldots\ldots\ldots, r_n$ be the collection of runlengths obtained after scanning the first plane. All the cells storing white runlengths will store the x and the y coordinates of the first pixel of that cell and the cell with black runlengths will store the corresponding depths of the pixels. In the

first plane the depth for all the pixels is zero and it increases as we go inside the image. The whole structure representing the complete information iis as shown,



Since all the planes are parallel to each other, the (x,y) coordinates for a particular point in all the planes will have the same value except for the z-coordinate.The area which is black in the first plane will hide all of the area behind of it, hence there is no need to process the pixels which are behind. But the area which is white in the first plane does not hide other black areas behind it. The algorithm now will proceed by considering these white areas to see whether other areas of a scene are visible from the view point or not.

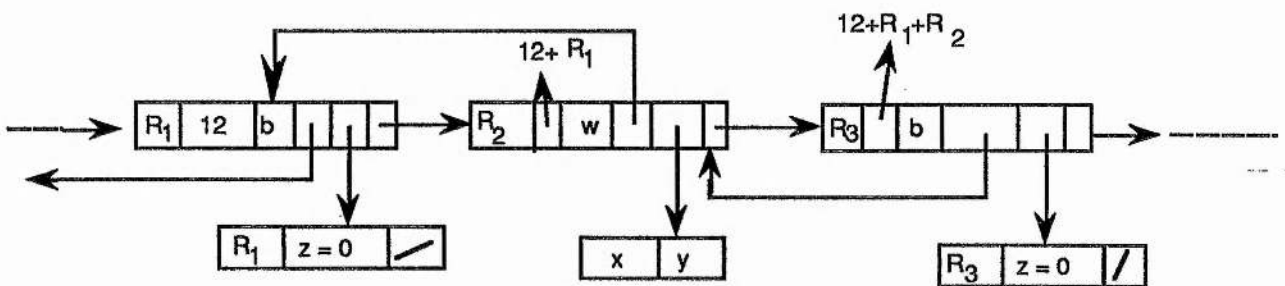
The information stored in each white cell of a linked list is,

1. the number of pixels (say *m* pixels) i.e. *runlength,*
2. the number of pixels used before a cell i.e., *Sum,* and,
3. the x and the y coordinates of the starting point.

Since all the planes are parallel to each other, the starting point for the next plane will be the same as the one obtained for the previous plane. We will start from that point and for the *m* pixels obtained from the white cell we will check the colour in the next plane. If all the pixels in the next plane are

white then we will consider the next white cell in the list and similarly will scan the next white area . But if some of the pixels are black then we will adjust our present linked list by turning those pixels to black. If the starting pixels or the end pixels of the *m* pixels, are turned to black then we will add these pixels to the left or to the right cell which is black, storing the depth information in the middle pointer. This can be summarised as follows.

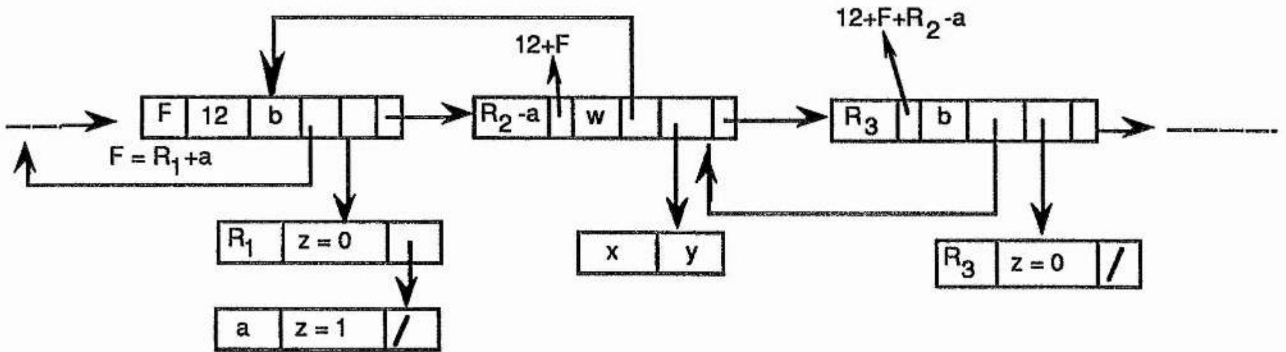
Let the white cell under consideration have runlength R_2 and let (x,y) be the starting point as below. We will scan the next plane (i.e., the 2nd) for that area starting from the point (x,y).



There are three cases,

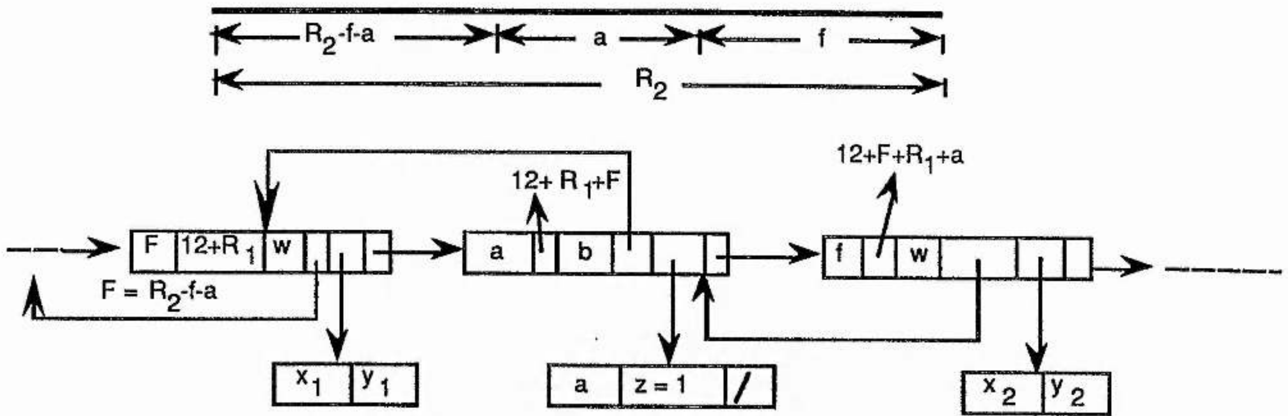
1. All the pixels in the 2nd plane are white. The linked list will be the same. Consider the next white runlength in the list, and scan this white area in the same plane and so on until we reach the end of the linked list. Our next step is to consider the 3rd plane and for each white cell in the modified linked list (*Note : if all the white pixels in the initial list are also white in the 2nd plane then the linked list will be the same otherwise we will modify it according to the position of the black pixels. Refer case 2 and case 3)*, scan the area in the 3rd plane and so on until the last plane is encountered.

2. Some of the starting pixels (say a) or end pixels are black in the 2nd plane. If a few starting pixels are black then we will add these pixel to the black cell which is to the left of the white cell under consideration, see below. If end pixels are black then we will add that information to the right cell. The new linked list, assuming the few starting pixels are black will be given as,

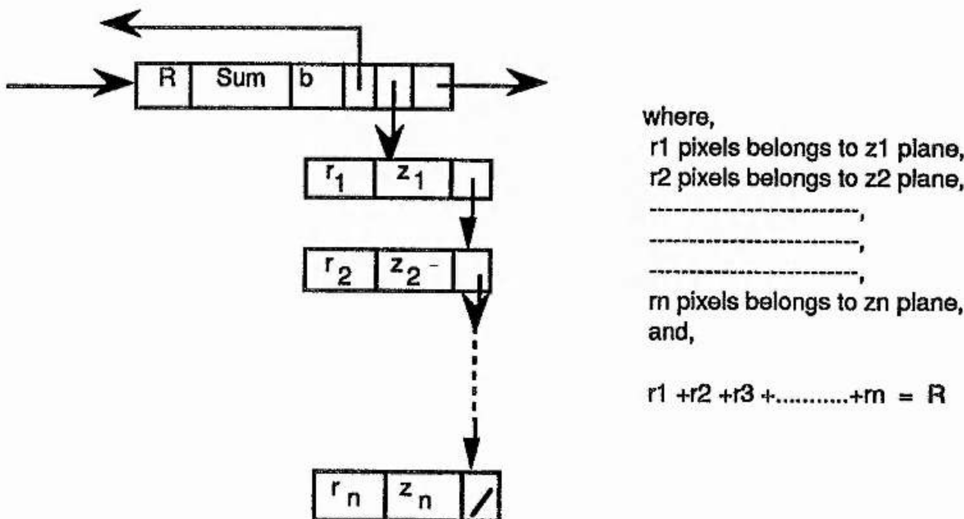


Now we will consider the next white cell and we will scan the same plane and so on until we will reach the end of the list.

3. A few pixels (say a) which are in the middle are black in the 2nd plane. Here we will divide R_2 pixels into three cells i.e., we have to add two more cells to the present linked list. The new linked list is as shown,



Once we have used a white cell, we will consider the next cell to find the visible area. All the planes will be scanned for a given white area in the previous plane. When all the planes have been processed for determining the visible area, we will get the collection of runlengths with hidden surface removed. Here a 3D.image has been transformed into a 2D-image. The middle pointer in each black cell will contain the depth information corresponding to the pixels in a cell. A typical black cell with depth information is as shown,

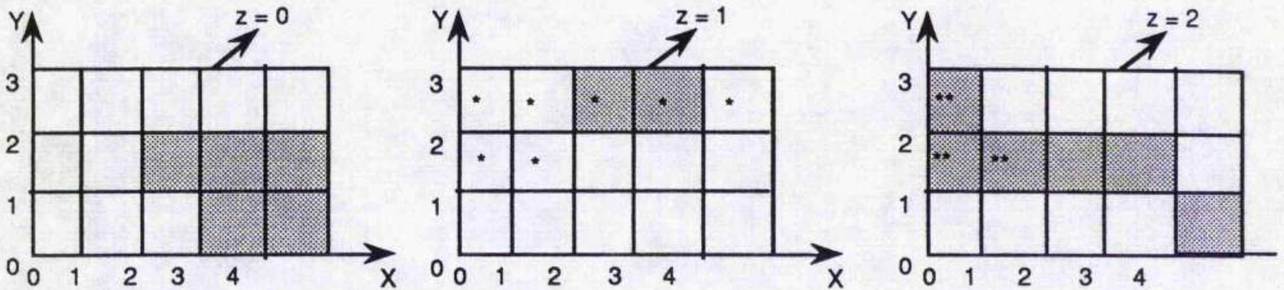


where,
r1 pixels belongs to z1 plane,
r2 pixels belongs to z2 plane,
------------------------,
------------------------,
------------------------,
rn pixels belongs to zn plane,
and,

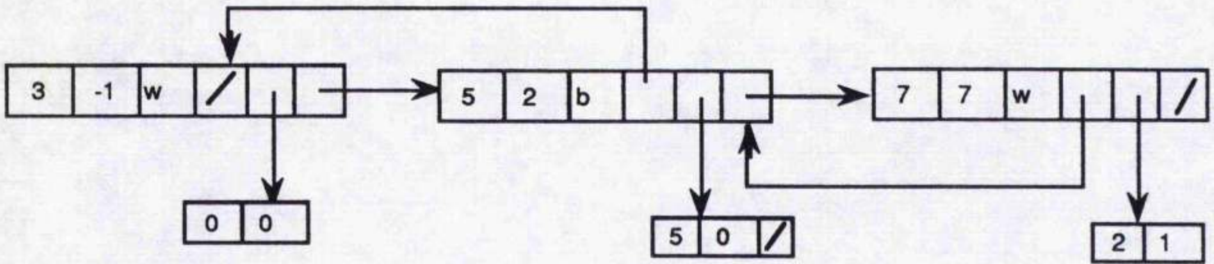
r1 +r2 +r3 +..........+rn = R

An example is given to explain the method discussed above.

Let us suppose, a 3D-image is composed of three planes each of size 5*3. The three planes with corresponding depths are given below,

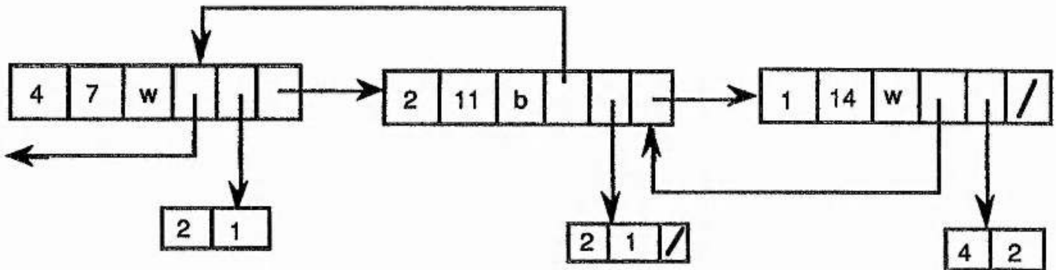


We will scan the first plane i.e. z=0, using a murray scan having radices r_2 r_1 equal to 3, 5, where r_1 belongs to x-part and r_2 belongs to the y-part. The corresponding linked list obtained is given below,

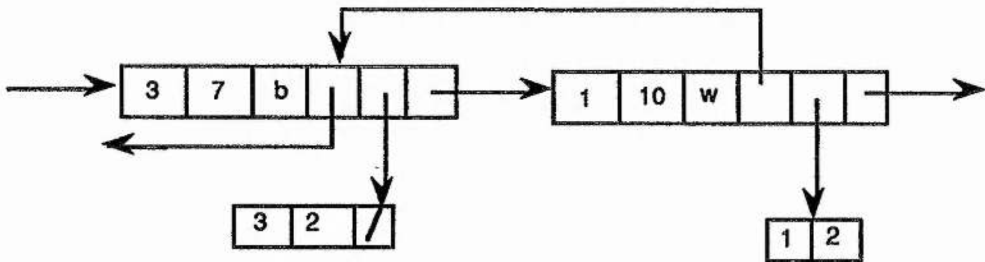


Now we will consider this list to find the visible area. Visibility is only possible where we have a white area. We will scan the linked list given above and for each white cell we will find the visible area. The first cell in the linked list is white with (0,0) be the start point and the runlength equal to 3. We will now consider the 2nd plane i.e., z=1, and will check whether any of these 3 pixels are black in this plane. The answer is no. The second cell which is white has a runlength of 7 and the coordinates for the starting point are

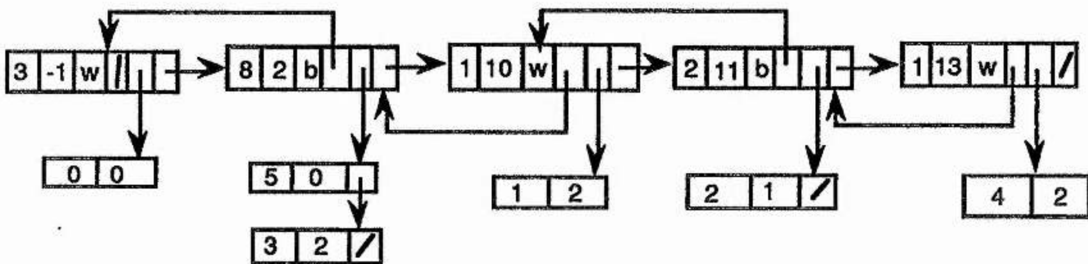
(2,1). In the 2nd plane, out of 7 pixels (marked *) two of them are black. We therefore change the linked list, as below,



Since the end of the linked list is encountered we will now consider this modified list and the next plane i.e z=2, to find the visible area. Consider all the white cells in that list, the white cell which has a runlength of 4 has 3 pixels black(marked **) in this plane.The new change in the list will now be,



We do the same with the other white cells if any. The final list will be given as,



7-3.3 Comparison of Hidden-surface Methods :

There are many different hidden-surface algorithms. Some use mathematically defined scenes and some use any arbitrary scenes. All have their own advantages and disadvantages. This has been discussed in detail in section 7-2.

The effectiveness of a hidden-surface method depends upon the characteristics of a particular application. If the surfaces in a scene are spread out in the z-direction so that there is very little overlap in depth, then a depth sorting method may be the best. Similarly if the surfaces are well separated in the x-direction, then a scan line or area subdivision method may be the best one. Depth sorting methods are very effective if a scene has a few surfaces and only a few of these are overlapping.

However if a scene has a large number of surfaces then the depth sorting method will require more memory than other methods. In this case octrees or area subdivision methods may be better. The method 2 discussed above is very similar to the octree approach. We do not need all the planes at the same time. We can process the planes one by one. The processing time depends upon the number of white pixels in the first plane. To obtain an inside view of a scene parallel to the XY-plane, we can start from any plane which is inside(say nth plane) such that $0 < n <$ no.of.planes. The advantage over an octree is again the size. In the case of an octree an image is a cube of side 2^n but with murray scans there is no restriction on the size of an image. Further as in octrees, we do not need to perform any sorting and intersection calculations in our method. The algorithm is the same for all the surfaces, no special considerations being given to the curved surfaces. Further if quadtrees or octrees have an advantage then this can be reproduced in murray scans.

7-4 Shading

7-4.1 Introduction:

One aspect of computer graphics that has received much attention is the production of shaded images of objects. Three types of calculations must be performed for producing computer generated images of three dimensional objects.

1) The hidden surface problem, which determines the visibility of a point from the viewpoint.This has discussed above in detailed.

2) The normal vector to the object at those points which are visible. This normal vector is used to determine the pixel or point color, for portraying smoothly shaded surface.

3) Intensity calculations. This takes the factors such as, the unit normal vector at that point, the direction of the light sources, the location of the observer, and the characteristics of the surfaces to derive a function that determines the proper intensity for the corresponding point on the image.

The simplest of these functions considers that the surface of the object has diffuse reflection i.e., the light that strikes the object from a particular direction will reflect off in all directions. Since the reflections from the surface are scattered in all directions, this implies that the location of the observer does not affect the intensity of a surface. The only calculation then is to find the angle between the light source and the normal vector to the surface. Usually the intensity of a pixel is directly proportional to the cosine of the above angle. The cosine of the angle can be determined by computing the dot product of the two unit vectors. If the dot product is negative, it means

that the viewer is on the opposite side of the surface from the light source. In this case the intensity should be set to zero. Distances are ignored, since both the light source and the observer are assumed to be at infinity from the surface. Further simplification can be made by assuming only a single light and the position of the observer is on the z axis of the coordinate system. The intensity functions are then proportional to the z-component of the unit normal vector (Figure 7.11) . Gouraud(1971) used this formulation for a very rapid smooth shading system.

In addition, there is a small amount of light which falls on the surface uniformly from all directions. This is known as ambient light. This constant should also be added to the intensity. The net diffuse reflection for a surface illuminated by ambient light and one light source is:

$$d = \max[0, N.L]$$

$$i = kd\,[Ia + d.Ip]$$

where

i = Perceived intensity.

kd = Coefficient of reflection for the surface. It lies between 0 and 1.

Ia = Incident ambient light intensity.

Ip = Intensity of the source.

d = Amount of diffuse reflection.

N = Normal vector to surface.

L = Light direction vector.

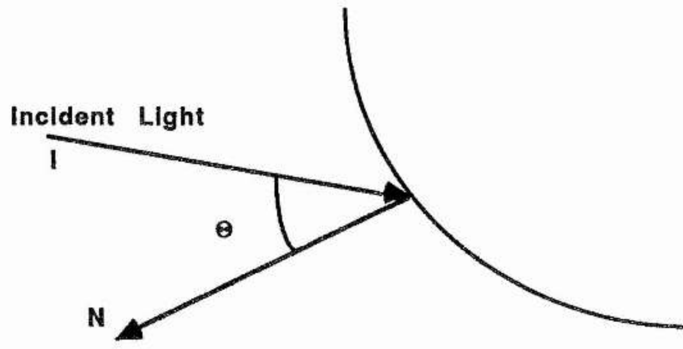
This model is simple to compute and quite adequate for many applications. The next section examines several shading models in detail.

7-4.2 Surface-Shading Methods :

The shading functions assume that a normal vector corresponding to the surface direction at the point being shaded is known.The accuracy of the normal vector representing the actual surface depends on the method used to obtain it.

The easiest way of obtaining a shaded image of a curved surface is to approximate it by many planar polygons. When two or more two dimensional images are used to obtain a three dimensional image, the surface description is easily available. For a mathematically defined surface, these polygons may be obtained directly from it. Since planar polygons are defined by a single surface normal, this information can be used in a simple shading function to determine the color for a whole polygon. This technique is limited. The resulting image will appear faceted although it may be possible to obtain a reasonable impression of the surface structure.

Gouraud attempted to improve the impression of a curved surface by smoothing over the polygonal mesh. His algorithm requires specification of the normal vector at each vertex of each polygon. This vector can either be obtained directly from the mathematical description or by averaging the normals of all the polygons adjacent to a vertex. Once a color has been obtained for each vertex, the color can also be obtained for any edge by interpolating between the endpoints color.These edge colors are linearly interpolated along a scan line to obtain the interior colors. Figure 7.12 demonstrates this interpolation scheme. Here the intensity at point 4 is obtained by interpolating the intensity values at 1 and 3. Similarly intensity at point 6 is interpolated from the intensity values at the vertices 1 and 2.



Intensity α cosΘ

Figure 7.11. The diffuse reflection component is proportional to the cosine of the angle between the incident light vector (I) and the normal (N).

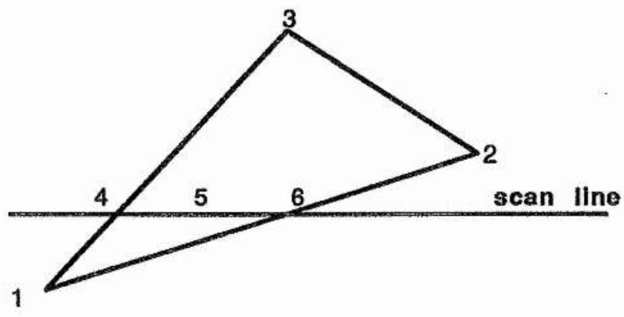


Figure 7.12. For interpolation shading, the intensity value at point 4 determined from the intensity values at points 1 and 3, intensity at point 6 is determined from values at points 2 and 1, and intensities at points(such as 5) along the scan line are interpolated between values at points 4 and 6.

obtained by interpolating the intensity values at 1 and 3. Similarly intensity at point 6 is interpolated from the intensity values at the vertices 1 and 2. Once the bounding intensities are found for a scan line, an interior point (such as point 5) is interpolated from the bounding intensities at point 4 and 6. This process is repeated for each scan line passing through the polygon. The result is a smoothly shaded surface that appears very much like the original curved surface.

Phong(1975) studied the physical properties of real surfaces and attempted to devise a function that produce more realistics results. In particular, he noted that, in addition to diffuse reflection, most surfaces specularly reflect some light. The amount of light reflected from the surface depends on the location of the observer. This phenomenon can be represented by a function which compares the angle between the direction at which light reflects off a surface and the direction in which the observer is looking; usually along the z axis (Figure 7.13). Since such light must reflect almost directly at the observer in order to be seen, Phong using the cosine of the above angle raised to some power, insured that it is significant at only very small angles. The exact value of the exponent is influenced by the surface reflectance properties. The surfaces can then be described by some combination of both the diffuse and specular functions plus an additional constant for the background illumination.

Gouraud shading removes the intensity discontinuities associated with the constant shading model, but still a number of problems arose with this simple interpolation scheme. The most serious problem was Mach band effect. Mach established the following principle :

Whenever the light-intensity curve of an illuminated surface has a concave or convex flection with respect to the axis of the abscissa, that particular place appears brighter or darker, respectively, than its surroundings [E. Mach(1865)].

This effect appears whenever the slope of the light intensity curve changes i.e. non-continuous first derivative. Phong reduces that effect by interpolating the normal vectors themselves between vertices and edges and computing a color for each pixel. This approach provides a more consistent surface definition, independent of the orientation of the polygons. It provides a smoother interpolation across polygons, although it is still not continuous in the first derivative.

When rotation is applied to the images, both Gouraud and Phong shading shows difficulties, with the shading varying significantly from frame to frame. This effect is due to the shading rule which is not invariant with respect to rotation. Consequently as the orientation of an image change from frame to frame does so the color.

Torrance and Sparrow(1967) present a theoretical model for reflecting light. Blinn(1977), Cook(1982), and Cook and Torrance(1982) used this model to produce synthetic images. The Torrance-Sparrow model for reflection from a rough surface is based on the principles of geometric optics. The surface of an object is assumed to be composed of many mirror like microfacets. The specular component of the reflected light is assumed to come from those that are oriented in the correct direction. The diffuse component comes from multiple reflection between facets and from internal scattering. The specular reflection for Torrance-Sparrow model is :

$$S = DGF/(N.E)$$

where,

D is the distribution fountain of the directions of the micro facets on the surface.

G is a geometric factor due to shadowing and masking of one microfacet by another.

and F is the Fresnel reflection law.

Torrance-Sparrow assume that the microfacet distribution on the surface is Gaussian, given as :

$$D = c_1 \exp(-(d/m)^2)$$

where,

c_1 is an arbitrary constant

m is the root mean square slope of the microfacets.

d is the angle between the normal to the surface and the normal to the microfacet.

Cook and Torrance use a more theoretically founded distribution model proposed by Beckmann, and Spizzichino(1963). The Beckmann distribution is:

$$D = (1/m^2 \cos^4 d).\ \exp(-(\tan d/m)^2)$$

which gives the absolute magnitude of the distribution function without arbitrary constants. Corresponding to specular reflection, if m is small then there is little difference between the Gaussian, Beckmann, or Phong

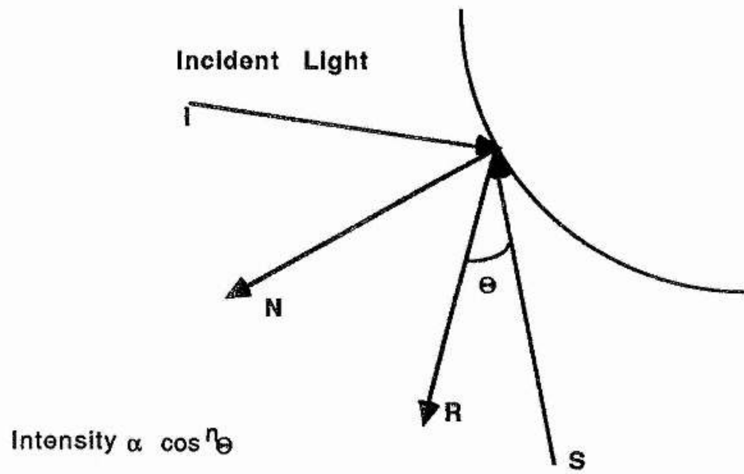


Figure 7.13. The specular reflection component is proportional to the cosine of the angle between the reflected vector (R) and the sight vector (S), raised to power (n).

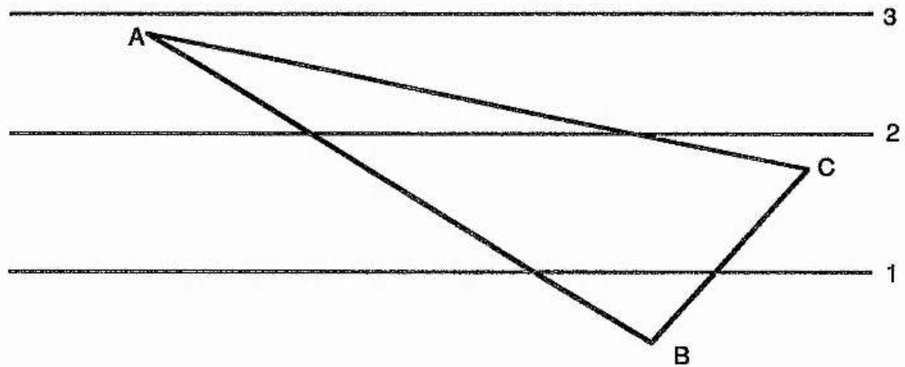


Figure 7.14. Each scanline has a list which contains the informations about edges which first become active on the scanline. For the above example, the list are as follows; list 1: AB, BC; list 2: AC; list 3: empty.

distribution functions(refer Roger(1985)). For larger values of m the differences are more significant.

Blinn(1977) compares the shape of the specular highlights obtained using the Phong illumination model, when the object is edge-lit. Edge lighting occurs when the angle between the observer and the light source is approximately 90^0. It yields very little improvement along the edge of the curved surfaces. If observer and the light source are at the same location, then the results for both the models are nearly the same.

When dealing with a mathematical representation of a surface such as bi-cubic patch, a difficult problem is determining the correspondence between a pixel in the plane and a point on the three-dimensional surface. For this reason, patches are generally approximated with a polygonal mesh and for shading we use the methods previously described. Catmull(1974) developed an algorithm which handled patches directly. His method recursively subdivides each patch into smaller sections until each covers only one pixel center. Blinn(1978a) and Whitted(1978) proposed algorithms that process patches with a scan line approach. Griffith(1984) presented a scanline algorithm, which generates a realistic picture of a solid object bounded by curved surfaces. Every scanline has a list of information about edges which first became active on the scanline (see Figure 7.14). Color finding is the same as given by Gouraud. Griffith pointed out that a disadvantage of this approach is that it is harder to tell whether a point is visible when the problem is 2D that it is when the problem is 1D.

7-4.3 Transparency :

Most of the illumination models and hidden surface algorithms assume that the object is opaque. When a transparent object is modelled then the intensity from the light source behind the object should be included in the

illumination model. Light passing through a surface is called transmitted light or refracted light. When a light is incident upon a transparent surface e.g. glass, part of it is reflected and part of it is refracted (see Figure 7.15).

Since the speed of light is different in different materials, the path of the refracted light will be different from the incident light, according to Snell's law[Brown(1955)] of refraction which states that *"for a given pair of media, the ratio of the sine of angle of incidence to the sine of angle of refraction is a constant, independent of the angle of incidence".* This law can be written mathematically as:

$$\sin Q / \sin Q' = n_2 / n_1$$

where,

n_1 is the index of refraction for first surface, Q is the angle of incidence,

n_2 is the index of refraction for the second surface and Q' is the angle of refraction.

Finding the refracted light using Snell's law is very time consuming. The refraction effect can be gained by simply shifting the path of the incident light by a small amount.

A simpler algorithm for the transparency effect ignores refraction. This approach assumes that there is no change in the index of refraction from one material to another i.e., the angle of incidence is same as the angle of refraction. This method can speed up the calculation of intensities. Newell,Newell and Sancha(1972b) proposed an algorithm for producing transparency effects. Here the intensity of a background objects is added to

the intensity of the transparent object. The intensity is then a linear combination of the two objects intensity given as:

$$I = rI_t + (1-r)I_b \qquad 0 <= r <= 1$$

where

I_t = intensity of transparent object.

I_b = intensity of background object.

and r = refraction coefficient just to weight the reflected and refracted intensity contribution.

The linear combination of the two intensities is not sufficient for modelling curved surfaces. Since the thickness of the material reduces at the edges, this reduces its transparency, as pointed out by Roger. Scott(1979) suggested a single nonlinear approximation based on the z-component of the surface normal, to find the refraction coefficient. The refraction coefficient is:

$$r = r_{min} + (r_{max} - r_{min})[1-(1-|n_z|)^r]$$

where

r_{min} and r_{max} are the minimum and maximum transparencies for the object,

n_z is the z component of the unit normal to the surface, and

r is a transparency power factor.

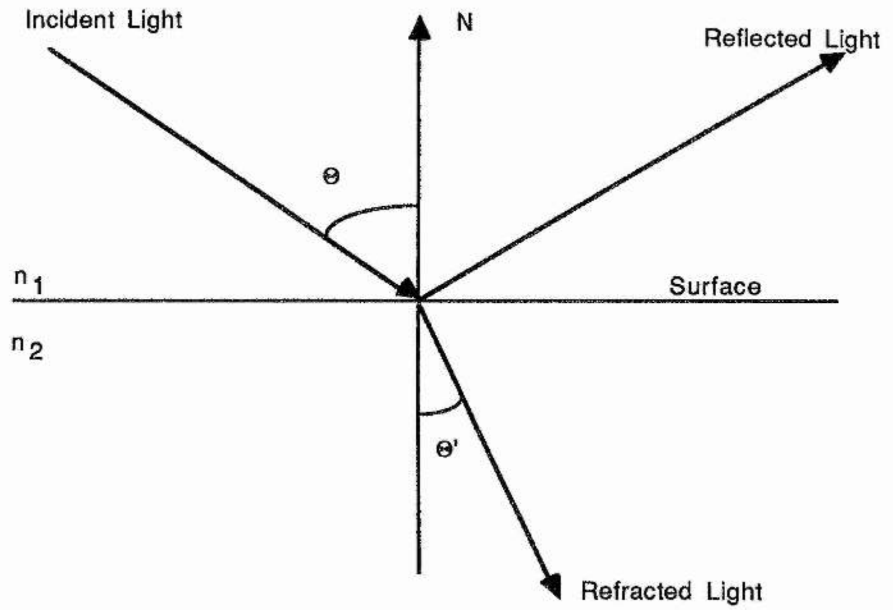


Figure 7.15. A ray of light upon a surface is partially reflected and partially refracted.

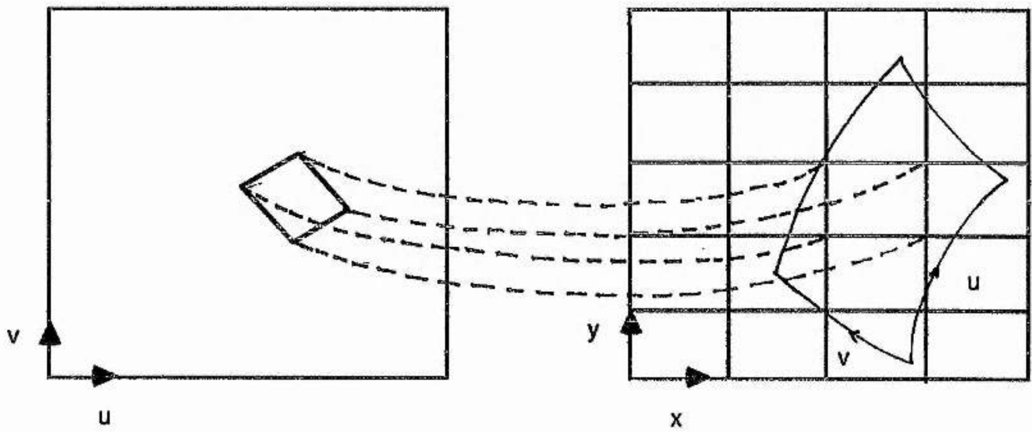


Figure 7.16. The area covered by a single pixel in xy space is mapped to its corresponding area in uv space. The texture pattern is then sampled from the uv mapping.

7-4.4 Texture Mapping :

In computer graphics, the texture is nothing but the surface detail in an object. Two type of texture are generally considered :

1) The addition of a separately defined pattern to a smooth surface.

2) Adding the appearance of roughness to the surface.

Blinn and Newell(1976) describe an addition to the Catmull algorithm (1974a, and b) and simulated this by mapping images onto surface patches. Each patch is associated with a particular stored image corresponding to the patch definition in parametric space. When the area on the patch that corresponds to a single pixel on the display is determined, the associated area in the image is mapped at that point and a color is calculated for this pixel (see Figure 7.16) .

If we want to add the appearance of roughness to a surface, a photograph of a rough textured pattern could be digitised and mapped to the surface. As Roger pointed out the results obtained are unsatisfactory because they look like rough-textured pattern painted on a smooth surface. According to him the reason for that is, the true rough-textured surfaces have a small random component in the surface normal and hence in the light reflection direction. Blinn(1978b) noted that problem and developed a method for perturbing the surface normal.

7-4.5 Antialiasing :

The nature of raster scan graphics is such that a color must be determined for a fixed number of dots in the horizontal and vertical direction. This sometimes results in distortions being introduced into computer generated images, known as aliasing. This is purely due to the results of this

digitization process. Here each dot or pixel is addressed by an integer pair (x,y). If in object space a point has been represented by a real pair then it has to be truncated to get an integer pair, before addressing it on the screen. Crow(1977) studied this problem in detail and proposed a number of solutions.

One of the most obvious problems involves straight borders that appear jagged. It is because the transitions between scan lines are discrete. This can be adjusted either by adjusting pixel positions (pixel phasing) or by setting the pixel intensities according to the percent of pixel area coverage at each point. Filtering can also be used to reduce aliasing effects.

7-4.6 Shadows :

Whenever a computer generated image of an environment is created such that a light source location is different from that of the observer, shadows appear. If the observer and the light source are at the same point then no shadows will appear. As Roger cited a shadow consists of two parts: an umbra and a penumbra. The central dense shadow area which is black is the umbra and the lighter area surrounding the umbra is called the penumbra.

Due to the algorithmic and computational difficulties only shadow umbra is included in creating computer generated images. The shadow calculations depend upon the location of the light source. To add shadows to a scene the hidden surface problem will be solved twice, once for the position of the light source and once for the observer's position. The object can be viewed from in front, above and to the right. Generally two type of shadows are considered: self-shadow and projected-shadows. Self-shadows result when the object itself prevents light from reaching some of its plane. A projected-shadow results when an object prevents light from reaching another object in the scene. The shadows depend only on the position of light source and not on that of the observer.

Atherton, Weiler, and Greenberg(1978) have extended the hidden surface algorithm, based on the Weiler-Atherton(1977) clipping algorithm to include shadows. The algorithm works in object space. Hence, the result can be used for accurate calculations. They employ a hidden surface process that produces a list of visible polygons as its output. A shadowed image is produced by first determining which surfaces each light source can see. The illuminated surfaces are then applied as lighted details to the corresponding polygons in the original object description. A hidden surface view can then be generated from any observer position. Surfaces which are shadowed from the light source by other objects will automatically be rendered black.

An algorithm by Williams(1978) uses a z-buffer hidden surface process similarly to determine curved shadows cast on curved surfaces. In this case, an additional z-buffer is maintained indicating which surfaces cannot see the light source. A point by point transformation is then performed from the observer's point of view to determine which surfaces should be darkened.

7-5 Shading Using Murray Polygons :

As mentioned above, we need three types of calculation for producing a shaded image of a 3-dimensional object. They are,

1. Hidden-surface problem solution,
2. The normal vector at the visible point,
3. Intensity calculations.

The problem of the hidden-surface, which determines the visibility of a point from the view point has been discussed above in detail. In the following sections we will discuss in detail the remaining two calculations.

7-5.1 Determining The Surface Normal :

If we have the detailed description of the surface then calculation of the surface normal is straight forward. Generally for many surfaces only a polygonal approximation is known or none at all. If the plane equation for each polygonal facet is known then the normal for each facet can be determined from the coefficient of the plane equation. For example, if the given plane equation is,

$$ax + by + cz + d = 0$$

then the coefficient of x,y,and z will give us a vector that is normal to the plane i.e.,

$$N = a\mathbf{i} + b\mathbf{j} + c\mathbf{k}$$ where **i,j,k** are unit vectors.

In our case, we are working with any arbitrary image, whose description is not known to us. In this case the normal at a point can be obtained by finding a plane passing through it . Since we have to find a plane equation we need at least three points. Three points (*Note : the points must not be in a straight line*) can usually be obtained by considering the two adjacent points to a point for which we have to find a normal. Once three points are available the normal can be obtained. For example,

let $A=(x_1, y_1, z_1)$, $B=(x_2, y_2, z_2)$ and $C=(x_3, y_3, z_3)$ be the three points. Using these three points we can find the plane and then the normal to it. Alternatively the normal to a plane can be obtained by finding the cross product of the two adjacent vectors at one of the vertices (say C).

$$N = \vec{AC} * \vec{BC} = \begin{vmatrix} i & j & k \\ (x_2 - x_1) & (y_2 - y_1) & (z_2 - z_1) \\ (x_3 - x_1) & (y_3 - y_1) & (z_3 - z_1) \end{vmatrix}$$

$N = li + mj + nk$

where,

$$l = (y_2 - y_1)(z_3 - z_1) - (y_3 - y_1)(z_2 - z_1),$$

$$m = (z_2 - z_1)(x_3 - x_1) - (x_2 - x_1)(z_3 - z_1),$$

$$n = (x_2 - x_1)(y_3 - y_1) - (x_3 - x_1)(y_2 - y_1).$$

If N has value equal to zero this implies the three points are in a straight line.

7-5.2 Determining The Intensity At A Point Using Murray Polygons:

In this section we will discuss the use of murray polygons in finding the intensity at a given point with a given depth. The algorithm takes a linked list which has been obtained after removing the hidden surface area. The shading model which is used supposes that the surface of the object has diffuse reflection i.e., light reflecting in all direction after striking the object. Total diffuse reflection for a surface illuminated by ambient light and one point source is given as (refer Hearn, and Baker(1986)),

$$I = K_d I_a + K_d I_p (N.L)/ (d+d_0) \quad \text{-----------------} \quad (1)$$

where,

I_a = uniform intensity from all directions.

Note : The Sun 3/60 has 8 bit planes per colour i.e. each group can generate 2^8 (=256) shades or intensities of red, green, or blue. The total colours can now be equal to 2^{24} (2^8 corresponding to each gun). To get gray scale levels i.e. between white and black, the intensity corresponding to each gun should be equal i.e. we can have 256 gray scale levels where 0 corresponds to dark i.e. black and 255 corresponds to full intensity i.e. white. We choose our ambient light to lie between 30 to 65. Actually I_a can take any value between 0 to 255. If the ambient light has too high value then the object will just appear very bright with no sign of any shading and similarly if it is too low.

K_d = the coefficient of reflection or reflectivity for the surface. It's value lie between 0 to 1, according to the reflecting properties of the surface. If the surface is very reflective then we can assume it to be 1.

I_a = The intensity of the light source, and we assume it to be equal to 255 (maximum intensity).

d = the distance between a point and a light source.

d_0 = a constant used to prevent the denominator from approaching zero.

Note : Since our light source is assumed to be at infinity,we ignore the distance factor $(d+d_0)$.

N.L = the angle between a unit normal vector N and the direction to the light source with a unit vector L (see Figure 7.1).

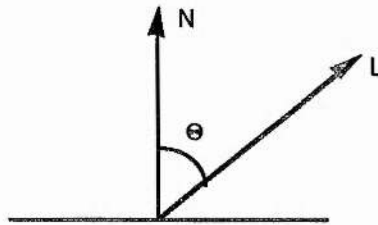


Figure 7.17.
Angle of incidence Θ between the light direction L and the surface normal N.

7-5.2.1 Determination Of The Angle Between N And L :

For each pixel which is black in an image, we have to find the intensity corresponding to the light source. Usually the intensity of a pixel is directly proportional to the cosine of the angle between the vector N and the vector L (N and L are defined above).

Let $A(x_1, y_1, z_1)$ and $B(S_x, S_y, S_z)$ be the two points, where the point A belongs to a black pixel in an image and point B belongs to the position of the light source. The unit vector L = AB can now be easily calculated.

Let V = a**i** +b**j** +c**k** be a vector,

where, $a = S_x - x_1$,

$b = S_y - y_1$,

$c = S_z - z_1$.

The magnitude of a vector V is given by,

$$|V| = \text{sqrt}(a^2 + b^2 + c^2)$$

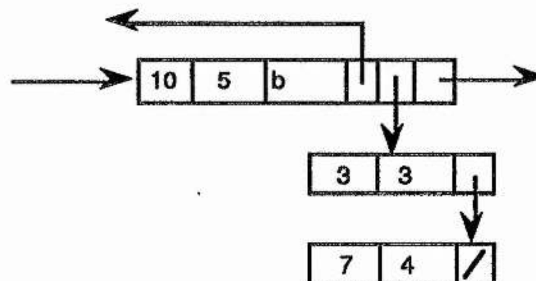
The unit vector L(say) having the direction of V can be obtained as,

$$L = V / |V| = li + mj + nk$$

where, $l = a/|V|$, $m = b/|V|$, $n = c/|V|$ and,

$$l*l + m*m + n*n = 1.$$

The source position $\bar{B}$ is known to us and the point $\bar{A}$ can be calculated using the information stored in the linked list. The linked list which we are using is obtained after removing the hidden-surface from a given view point. Using the item *'Sum'* stored in the list we can calculate which point it is on the curve and then the co-ordinates i.e. n --> (x,y). The corresponding depth can be obtained from the another structure which is linked to the list by the middle pointer. For example, a black cell which belongs to the linked list is given below,



Since the item *'Sum'* is given to be 5 and the item *'runlength'* is equal to 10, this implies that from the 6th to 15th points will lie in this cell and are black. Using the transformation from n ---> (x,y) discussed in chapter 2 we can calculate the coordinates for all points from the 6th to 15th. For the z-value we will scan the middle pointer. The information stored with the middle pointer states that, out of10 pixels, 3 pixels belongs to the 4th plane i.e. z = 3 and 7 pixels belongs to the 5th plane i.e., z = 4 (*Note : z = 0 is the first plane).* For example for the 6th point the co-ordinates will be given as (x,y,3). Now we have the source co-ordinates and the pixel coordinates , the unit vector L can be easily calculated using the above transformation.

The next step is to find a unit normal at a given point. The unit normal vector at a given point can obtained by finding the cross product of the two adjacent vectors at that point. The two adjacent point can easily be obtained directly from the murray scan. Let r_1 and r_2 be the first two radices i.e. the size of the smallest tile is $r_1{*}r_2$. These $r_1{*}r_2$ pixels are well packed in a small tile and are very near to each other. This coherence between the pixels can be used to find the two adjacent points. As discussed in chapter 2, a murray scan move forward either by incrementing or decrementing the x-value or the y-value i.e., by calculating the next co-ordinates. Hence the x and the y co-ordinates for the pixels can easily be obtained straight from the murray scan, which will pass through each and every point in a tile. The depth z corresponding to these pixel can then be obtained by scanning the middle pointer as explained above.

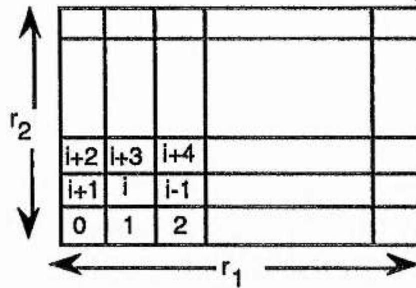
One can also use 4-point connectivity or 8-point connectivity to find the two neighbours, but the problem is to get the z-value. The points which we will get , may lie in the same tile or in different tiles. In any case, to get the z-value we have to find the nth point first then the cell in which it lies and the z-value corresponding to that point, i.e.,

f(x,y) ----> n

n ----> z-value

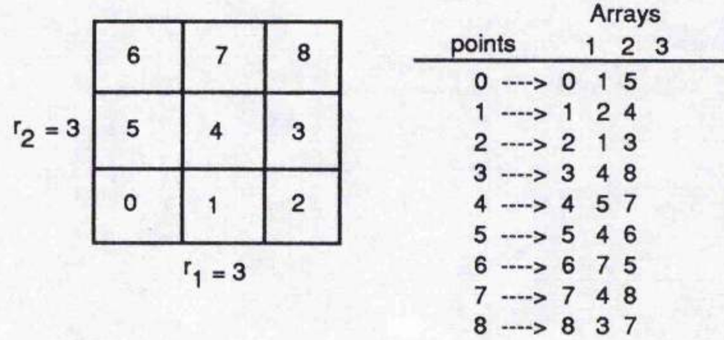
It may also be possible that the two neighbours are white. In this case again we have to consider two arbitrary neighbourhood points. This scheme is definitely going to consume more time than considering the consecutive points of a tile.

Consider a tile of size r_1*r_2 as shown,

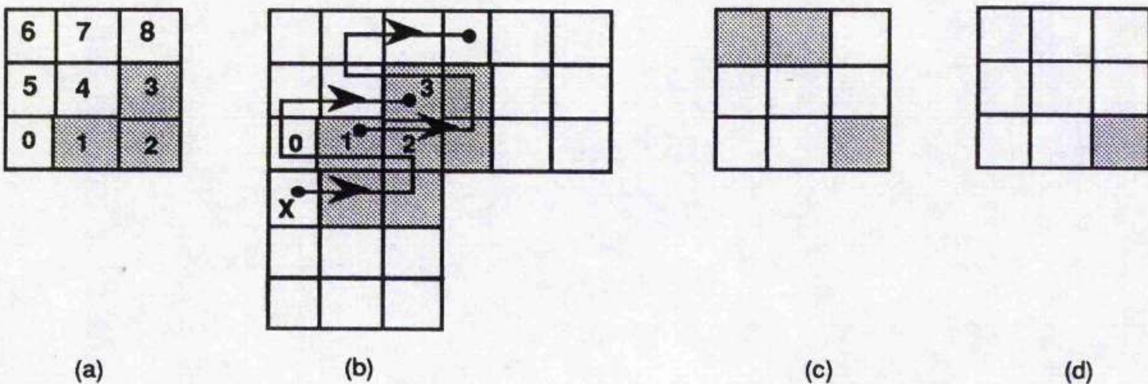


The pixel at position 0 has three neighbours, they are 1,i, and i+1. We can arbitrarily choose any two points out of these three neighbours. Similarly for the ith point which has eight neighbours, any two points can be selected. Since we have three points we can easily find the normal by finding the cross product between the two vectors passing through that point. The algorithm can be made faster if we pre-assign the two neighbouring points to each point in a tile. Here we need three array of integers, one to store the point number for which we have to find the normal and the other two corresponding to the neighbourhood points numbers. If the tile size is large then the array size will also be large and also the coherence between the first and the last pixels will be lost. So it is better to use the smallest tile size i.e., 3*3 (*Note : we can*

*also use 2*3 but a problem arises if we have to convert the scan for scaling).* The tile of size 3*3 and the neighbourhood points to each point in a tile is given below.

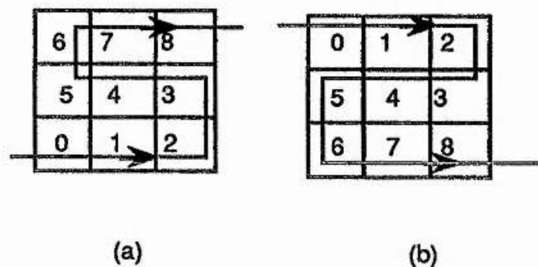


If all the pixels in a tile are black then we can easily get the two adjacent points as assigned above to find the normal. Problems can arise when some of the pixels are not black. This generally happens when we are dealing with the boundary points. For example, a tile where only three pixels are black is given below(case a),



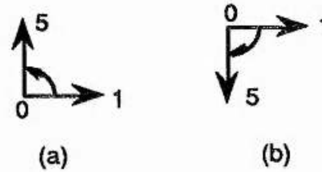
In case (a) the pixel marked 3, which is black has both assigned neighbours pixels i.e., 4th and 8th are white. Now one way of finding a normal at the pixel marked 3 is to use any two arbitrarily black pixels in the tile. Here we can use pixels marked 1 and 2. But if either the pixel marked 1 or the pixel marked 2 is white then there is no way to get two more black pixels in the same tile. To obtain other black pixels we can consider another tile which will start from a arbitrary point which is connected to the pixel under consideration and will contain that pixel i.e., 3rd. We can start our tile from the 1st pixel (see case (b)). This we do just to get more black surrounding pixels. Once we have three black pixels the normal can be easily calculated. There are many cases which can be solved similarly (see above case c and d).

Using the above information a unit normal at any given point can be easily calculated. The problem can arise with the direction of this unit vector. If it is pointing in the wrong direction then the shading corresponding to the point is wrong. This effect is due to the orientation of a murray scan from tile to tile. As we know that a murray scan has four possible orientations, which can effect the angle between the two adjacent vectors. For example two tiles of size 3*3, with two different pattern is given below. Since the polygon is entering from the two different sides, the numbering in both the tiles will be different.



(a) (b)

Consider the pixel at position zero in both cases. In case (a) the direction for the angle between the two vectors is anticlockwise where as in the second case (b) it is clockwise (see below), which will bring the change in the direction of the unit normal.



(a) (b)

To determine whether it is pointing in the right direction or not we can consider the dot product between the unit normal vector and the direction to the view point with unit vector V (say). If dot product is negative then the normal is pointing in the reverse direction. To make it point in the right direction we will multiply with the components of the normal vector by -1 i.e., if N = (L , M , N) be the initial unit normal vector then the correct normal vector will be given as N = (-L , -M , -N).

Now we have the unit normal vector at a given point(say N = (L , M, N)) and the unit vector towards the light source (say L = (l , m , n)). The dot product will now be given as ,

$$\cos \Theta = N . L = l*L + m*M + n*N$$

If cos(Θ) is in the interval from 0 to 1 then the surface will be illuminated by a point source. If cos(Θ) is negative, the light source is behind the surface.

Now we know the values for all the parameters in the illumination model, the intensity can be calculated for any given point. We have to calculate the angle of incidence i.e N . L, for each visible point in an image. The procedure for setting a colour map table is as given,

```
! This program sets up a vector (or color map table) of all the
! different planes
! Each different set of planes will have different gray scale value.
! For example off & off & off ... & off is black and so on.

let g = vector 0::255 of off
for I = 0 to 255 do
begin
   let d := I
   g(I) := { if d rem 2 = 0 then off else on}
   d := d div 2
   for j = 1 to 7 do
   begin
         g(I) := g(I) & { if d rem 2 = 0 then off else on}
         d := d div 2
   end
end
```

Initially the shading obtained on the edges and on the surface of a sphere was not very smooth. By examining the shade color it was noticed that some of the intensity values were either very high or very low, in comparison to the surrounding intensities. In order to obtain reasonable shading we decided to use smoothing techniques. In the next section the smoothing technique used for the noisy data and the results obtained is discussed.

7-5.3 Smoothing [Lanczos(1957), and Cole, and Davie(1969)] **Of Data :**

In this section we will discuss the smoothing of noisy data by making a least squares fit to a suitably chosen polynomial. The correction is made point by point. At each stage only a few selected points about the one currently under consideration are used. The expression which is presented here corresponds to 5 points i.e., the two neighbour on both sides plus the

here corresponds to 5 points i.e., the two neighbour on both sides plus the central point. The smoothing of data by fourth differences using a parabolic equation of the second order is discussed below,

A parabola of the second order is given by,

$$y = a + bx + cx^2 \quad \text{----------} \quad (1)$$

and the data belongs to the points $x = -2, -1, 0, 1, 2$. The aim is to combine every measurement with its two neighbour to the left and to the right. The problem is to minimise,

$$\Sigma (y - y_i)^2 \quad \text{-----------------} \quad (2)$$

with respect to a, b, and c. Our goal is to correct the central value y_0, which belongs to x=0. At x=0, the central value $y_0 = a$, so we solve the normal equations which are obtained after differentiating equation (2) with respect to a, b, and c, to find the corrected value for y_0. The corrected value obtained is equal to $y_0 - 3\delta^4 y_0/35$, where $\delta^4 y_0$ is the fourth central difference of the zero line. The above result is given in Lanczos(1957)

Once the central point has been smoothed we move down one step and then again consider 5 points for the next point to be smoothed. The corrected values for the first two observations and the last two observations are given below,

$$y_{-2}(\text{corrected}) = y_{-2} + \delta^3/5 + 3\delta^4/35$$

$$y_{-1}(\text{corrected}) = y_{-1} - 2\delta^3/5 - \delta^4/7$$

$$y_2(\text{corrected}) = y_2 - \delta^3/5 + 3\delta^4/35$$

$$y_1(\text{corrected}) = y_1 + 2\delta^3/5 - \delta^4/7$$

Here in each case we have to make a central difference table. We can avoid a central difference table by finding the coefficient to be multiplied by the data values selected. Consider a difference table given below,

x	f(x)	$\delta f(x)$	$\delta^2 f(x)$	$\delta^3 f(x)$	$\delta^4 f(x)$
1	x_1				
		x_2-x_1			
2	x_2		$x_3-2x_2+x_1$		
		x_3-x_2		$x_4-3x_3+3x_2-x_1$	
3	x_3		$x_4-2x_3+x_2$		$x_5-4x_4+6x_3-4x_2+x_1$
		x_4-x_3		$x_5-3x_4+3x_3-x_2$	
4	x_4		$x_5-2x_4+x_3$		
		x_5-x_4			
5	x_5				

Now $y_0 = y_0 - 3\,\delta^4 y_0/35$ or $x_3 = x_3 - 3\,\delta^4 x_3/35$, substituting the value for $\delta^4 x_3$ in the previous expression we will get,

$$x_3(\text{corrected}) = (-3x_1 + 12x_2 + 17x_3 + 12x_4 - 3x_5)/35$$

Now whenever we get five new points we will multiply them with the corresponding constants to give the smoothed value for y_0 (or y_3). Similarly the corrected values for the first two observations and the last two observations may be obtained and are given below,

$$x_1(\text{corrected}) = (31x_1 + 9x_2 - 3x_3 - 5x_4 + 3x_5)/35$$

$$x_2(\text{corrected}) = (9x_1 + 13x_2 + 12x_3 + 6x_4 - 5x_5)/35$$

$$x_4(\text{corrected}) = (-5x_1 + 6x_2 + 12x_3 - 22x_4 + 9x_5)/35$$

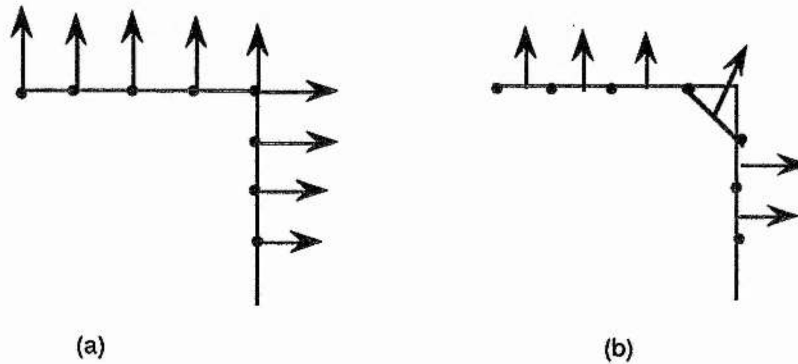
$$x_5(\text{corrected}) = (3x_1 - 5x_2 - 3x_3 + 9x_4 + 31x_5)/35$$

Similarly results can be obtained by considering seven points or nine points etcetera, with any suitably chosen polynomial. We can also use a cubic polynomial but for n =2m and n =2m+1, (where n is the degree of the polynomial and m is an integer), the correction factor is the same. With the linear case and considering three points i.e., one on each side, the correction factor obtained for all three observations under consideration is same after second iteration.

7-5.4 Results :

As discussed in section 7.2, the hidden surface algorithm transforms a 3D image into a 2D image by removing all the pixels which are hidden from a given view point. Those points which are visible are obtained from different depth planes. The depth of the pixels are numbered 0,1,2....., n-1, where n is the total number of the planes parallel to the XY-plane. The linked list obtained after removing the hidden-surface is used to shade the object. Our algorithm for generating shaded images accepts any arbitrary image. The image can be in any form, such as, a polygonal mesh, curved surface patches, or solid geometry construction. Since the scene is arbitrary we do not have any information about the object surface. If a scene happens to contain a cube then it is noticed that the shading on the edges is not very good. There is too much irregularity in the shade color. Since the scene is arbitrary we cannot easily detect an edge. In the case of mathematically defined objects the surface description is known to us and hence we can easily detect an edge . Since our images are arbitrary, hence we need to find a general solution which will give reasonably shaded images in all the cases. In the following paragraph initially we will discuss the reasons for these problems arising and in the end we will present some solutions to these problems

The reason for not getting good shading at the edges concerned the normal value, obtained at the edges. We will explain this by consider two planes meeting at an edge, as shown below,



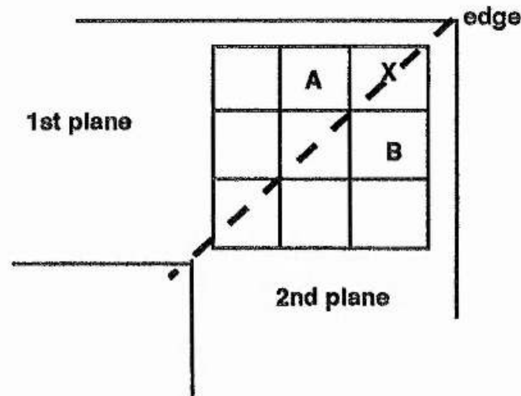
(a) (b)

We can consider two cases,

1. If we are using one point to find the normal (see case (a)) then there will be no problem at the edges. But theoretically it is not possible only if we precalculate the normal at each edges and then interpolate them(Phong(1975)), so that the normal at the edge between the two plane will then be one with less (or more) magnitude. In our case we find the normal at each point by considering two other adjacent points. This has been discussed above.

2. If we consider two points (or more) to find a normal (see case (b)) then at the edges some points can belong to one plane and others to the other plane, which will bring some distortion in the normals at the different edge points.

We consider three points to find a normal. A tile of size 3*3 and an edge passing through it is as given,



For all the edge points consider one point from the first plane and the other two from the other plane or vice versa. For example, the pixel marked X will take two pixels i.e. pixel marked A and B, to find the normal, where the pixel marked A lies in plane 1 and the pixel marked B lies in plane 2 and pixel X is present in both the planes, see above. Some improvement with the edges is discussed later.

Distortion is also seen in the case of a sphere. The shading is not very smooth. The reason for this was that the sphere was calculated explicitly from the equation

$$(x-a)^2 + (y-b)^2 + (z-c)^2 = r^2$$

giving, $$z = (+/-)\ [\mathrm{sqrt}(r^2 - (x-a)^2 - (y-b)^2)] + c$$

Here the x and the y values will be integers whereas the z-values can be real or integer. Since the planes are assigned an integer z-value, we have to truncate this value if it is not an integer. The result was not very pleasant

since the intensity values were fluctuating very much. The problem would not have arises with an image obtained by video camera.

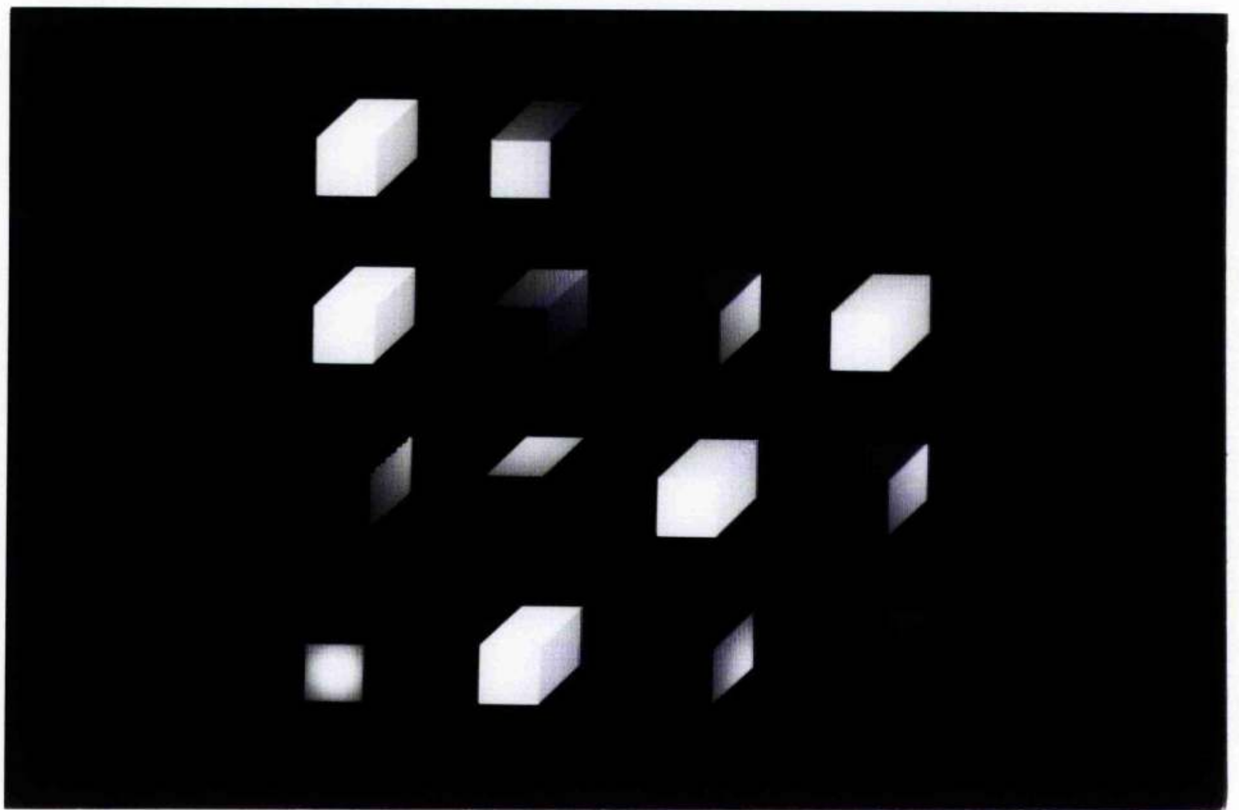
We tried to solve these problems by smoothing the data by using a parabolic equation of the second order as discussed above. The correction was made point by point. At each stage only a few selected points about the one currently under consideration were used. Initially we used 5 point smoothing i.e., two neighbour on both sides plus the central point. Our aim was to combine every measurement i.e., intensity, with its two neighbour to the left and to the right. The results obtained by this method were not very satisfactory. In the case of a sphere it appeared as if some one had pressed it from all sides. The edges were also jagged , see Figure 7.18. Since the points in a tile are very close to each other we tried to use seven point smoothing also. The method was repeated many times. The result was as shown in Figure 7.18. We also tried to smooth the dot.product instead of the intensity. We thought, since the dot product was between 0 and 1 then there was a chance of getting better smoothing than by using intensities which lie between 1 and 256. The result however was the same.

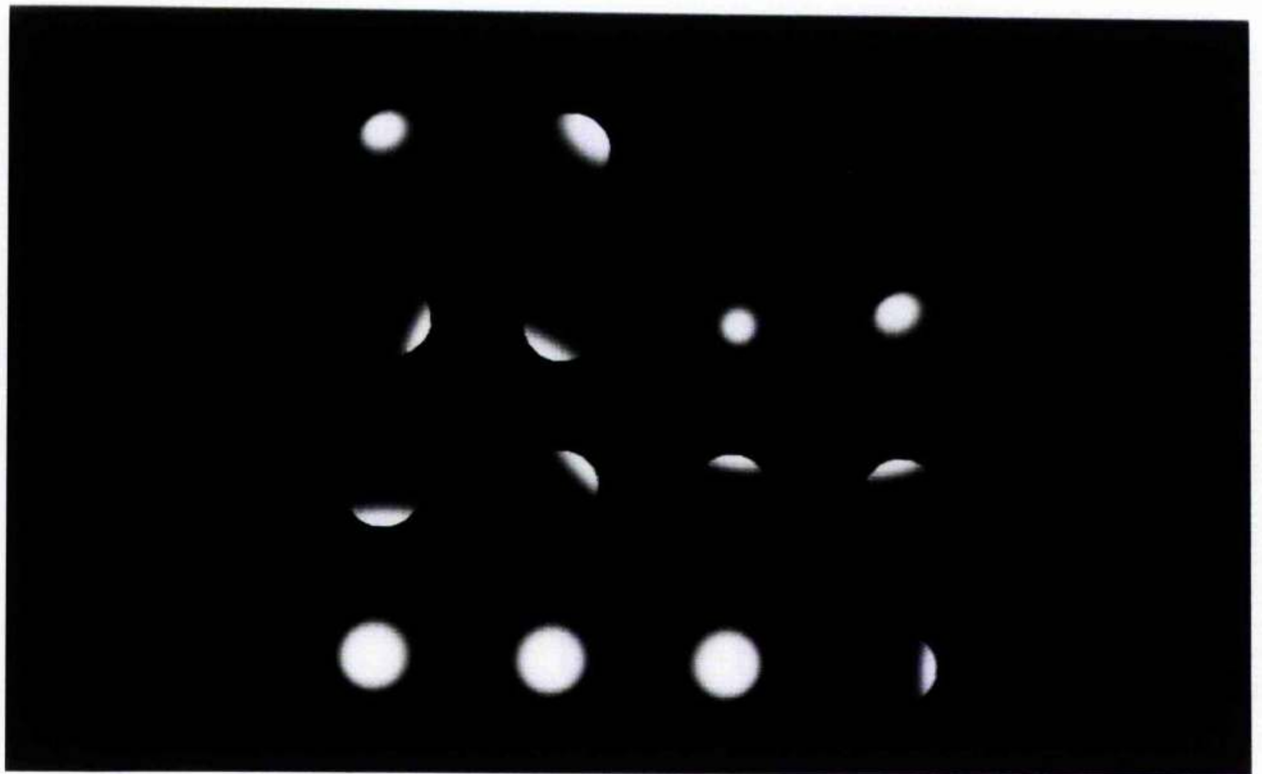
In the case of the edge problem the right solution was obtained as follows,

1. Most of the edge points will have 4 connected points, neglecting the diagonal points. At each point we find a normal to find the intensity. As we have seen the intensity at the edges is not very smooth, and the reason lies in the normal vector. We solved this problem by considering four different normals obtained by considering four different pairs, and then taking the average of these normals to give the approximate unit normal vector. We

Figure 7.18. Shaded picture of a cube. The position of the light source is changing, and the shading function assumes that the surface has diffuse reflection. The edge problem is also clearly shown.
Note: Cube has been obtained by putting the planes one after the other.

Figure 7.19. Shaded picture of a sphere. The position of the light source is changing, and the shading function assumes that the surface has diffuse reflection.





could also use diagonal points to increase the number of normals, but this makes the algorithm very slow.

2. Finding four normals at each point of an edge can be very time consuming. We can alternatively find only two normals vectors and consider the one which is either small in magnitude or large. If we select the normal which is small in magnitude then in all the case we will consider the normal with small magnitude and vice versa.

3. Since the intensities at the edges will be different from the two adjacent planes and hence the intensity can be adjusted by comparing the neighbourhood intensities e.g. 113, 68, 219. The middle one can be replaced by either 113 or by 219. If we select 113 then from then on we will consider the one which is less and vice versa. But this approximation is not always satisfactory, for example, two planes adjacent to each other, having a small gap which is one pixel wide will always have this gap black.

In case of the sphere the appropriate shading was obtained when we used the real z-value for all the points for which we stored the truncated z-values in the list(*Note :* In general, this is all we have). The result are shown in Figure 7.19. This however is graphics rather than image processing.

Conclusion:

From the above results and discussion we have noticed that the depth value corresponding to pixels plays an important role in getting a reasonable shaded image. Hence if we are dealing with mathematically defined scenes then the correct z-value should be stored in order to get a reasonable shaded

image. But for an arbitrary image we don't have to worry about that since we have to use the data as presented. As long as an image is made up of curved surfaces a normal at each point can produce a satisfactory shaded image. But suppose our image is made up of curved surfaces and solid boxes, then some problems may arise at the surface intersections. To get reasonable shading at the edges we therefore have to use two normals at each points. Since the scene is arbitrary, it is necessary to compute two normals at each point. The one whose magnitude is less (or more) can be selected to find the intensity. Two normals for a point can be obtained straight from the tile which is under consideration.

7-6 Specular Reflection :

At certain viewing angles a shiny surface reflects all incident light, independent of the reflectivity value of the surface. The result is a bright spot of reflected light. This phenomenon is known as specular reflection. In case of a perfect reflector e.g., mirror, the angle of incidence and the angle of specular reflection are same. The complete intensity model for reflection due to ambient light, incident diffuse reflection, Phong specular reflection and a single point source can be written as,(refer Hearn, and Baker(1986)),

$$I = K_d I_a + I_p [K_d(N.L) + w(i,\lambda) (V.R)^n] / (d+d_0) \text{ -------- } (2)$$

where,

w(i,l) = It gives the ratio of the specular reflected light to the incident light as a function of the incident angle i and the wave length l. We can simplify the intensity calculations by setting w(i,l) to a constant value K_s for

the surface. Its value can lie between 0 and 1 depending upon the surface material.

V.R = angle between the direction at which light reflects off a surface and the direction that the observer is looking (Figure 7.20).

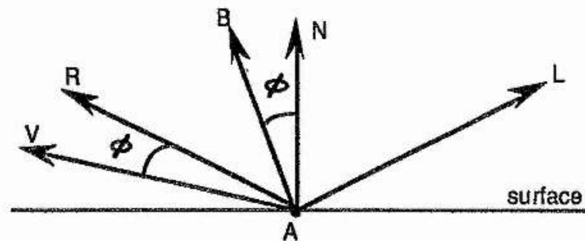


Figure 7.20. Angle between the direction at which light reflects off the surface (V) and the direction in which the observer (R) is looking.

n = determines the type of the surface to be viewed. For shiny surfaces n can take a value equal to 200 or more and for dull surfaces it's value can be one.

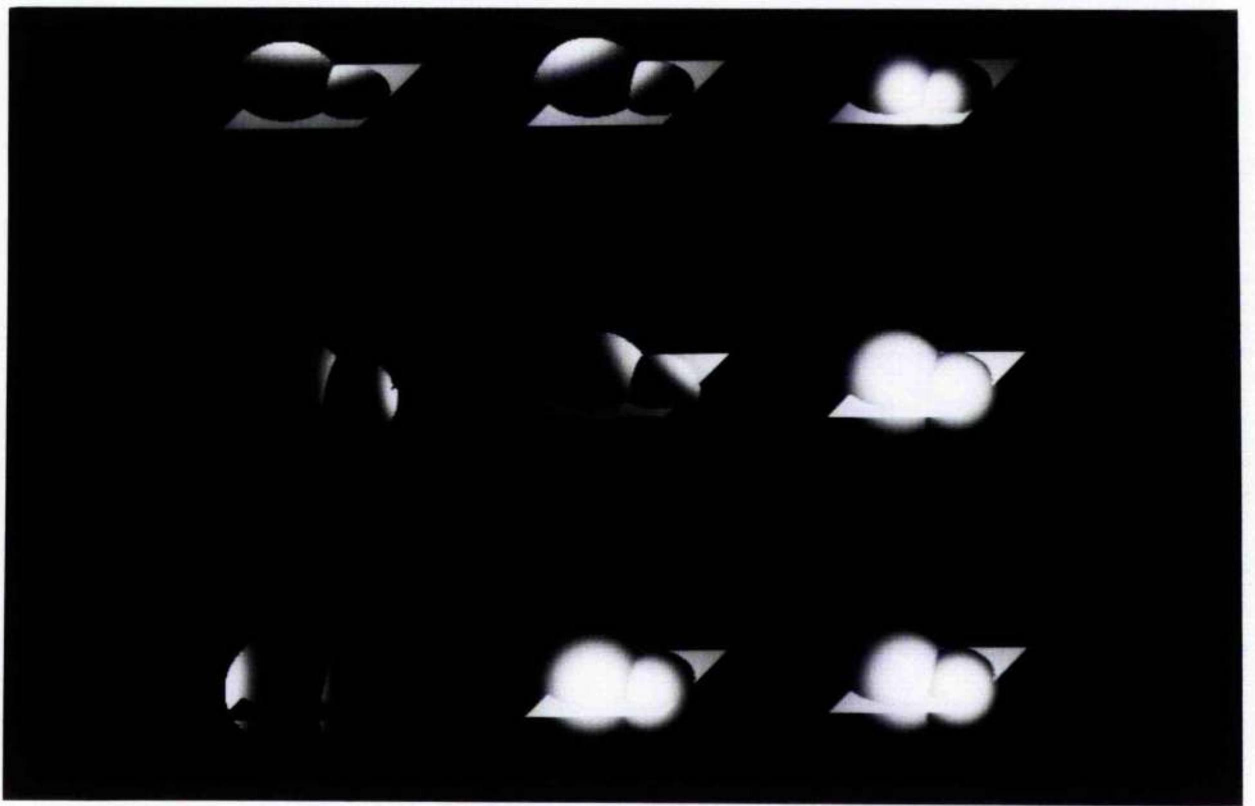
The angle between the vector V and R can easily be calculated. If we consider the Figure 7.20, the direction of the normal vector N bisects the angle between the vectors R and L. Similarly we can define another unit vector B(say) which will bisect the angle between the vectors V and L and will be given as,

$$B = (V+L) / |V+L|$$

therefore, $V.R = N.B$

Figure 7.20. Two intersecting spheres with a plane passing through them. The position of the light source is changing, and the shading function assumes that the surface has diffuse reflection.

Figure 7.21. Two intersecting spheres with a plane passing through them. The position of the light source is changing, and the shading function assumes that the surface has specular reflection.



Since the position relative to the light source (L) and to the viewer (V) is given, hence the vector V and vector L can easily be calculated as discussed above, and so the vector B.

The shading model given in equation (2) has been used to calculate the intensities for the black pixels in an image. In comparison to the diffuse reflection the specular reflection produces a bright white spot of reflected light. It has noticed that when the object is edge lit both the shading functions produces the similar result. The appearance of white bright spot when the object is not edge lit generates very realistic image. The result obtained from the shading models is shown in Figure 7.21.

7-7 Remarks :

In comparison to the octree and linear approaches, the murray approach is going to be slightly advantageous. As we know the intensity at a point in an image depends on the angle between the unit normal vector and the vector L towards the light source. Finding the vector L is very easy since we have the coordinates for both the points i.e one belonging to the light source and the other one belonging to the point in an image. The problem comes with the normal vector N. To find a normal at a point we need at least two more points . In case of linear encoding or quadtree (*Note : this quadtree is obtained after transforming an octree into a quadtree)* encoding, these two points can be calculated either by using 4-point connectivity or by 8-point connectivity of a point (i,j). The corresponding z-value can then be obtained from the linked list which stores the z-values for each visible black point. If the two computed adjacent points happen to be white then we have to repeat this process again. But in the case of a murray scan the connected points with x, y, and z-values can be obtained straight from the scan. We do not have to do extra work in

finding out the z-values corresponding to the connected points. Only at the boundary we have to do the calculations to get the z-value, as discussed above.

Phong shading will be computationally more expensive than the murray approach. The reason is the extra pre-processing step required. In the case of Phong shading, a curved surface(or any surface), initially it has to be approximated as a set of planar polygons and then at each vertex where polygons meet, a normal has to be determined. This vertex normal is then interpolated linearly across the polygon surface. But with the murray approach no special consideration needs to be given to the curved surfaces (or any surface); and also the complexity is proportional to the number of pixels and is independent of the number of faces. Similarly with Gouraud shading, where at each vertex we have to find the intensity, which is going to interpolate across the polygon surface, the computational time will be less in comparison to Phong but will be more in comparison to the murray approach.

Chapter 8

8. IMPLEMENTATION 244

8-1 User Interface 244

8-2 Menu Design 245

8-1 User Interface :

In designing a graphics package, two points should be considered

i. The graphics operations to be performed,

ii. How to present them to a user.

The interface should be designed in such a way so that it is very easy and efficient for the user to access basic graphics functions. The graphics package might be set to produce engineering design, business graphics, as an artist's paintbrush etcetera. A user interface generally considers the following components,

1. User model
2. Command language
3. Menu formats
4. Feedback methods
5. Output formats

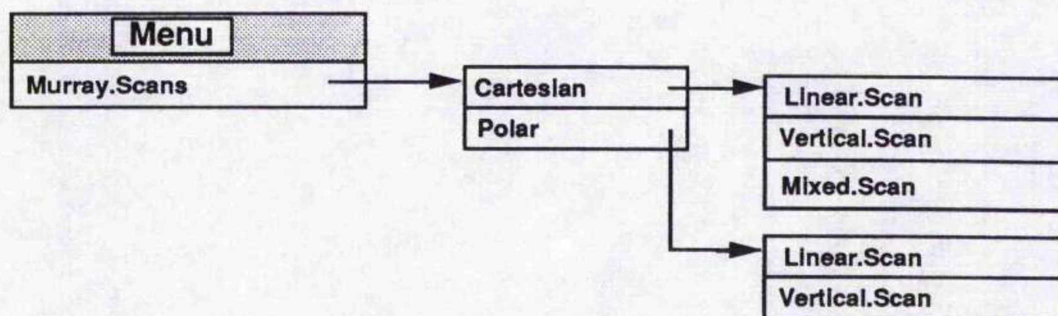
The package which we made contains only the menu format (or Test bench). All the processing options are present in the menu format. In the following sections we discuss the options(or operations) which are present in the menu and finally the whole scheme is shown in pictorial form. We made two packages; one using PS-algol and other one using C. Both the packages have the same options but with a difference in the processing time, this is discuss later on.

8-2 Menu Design:

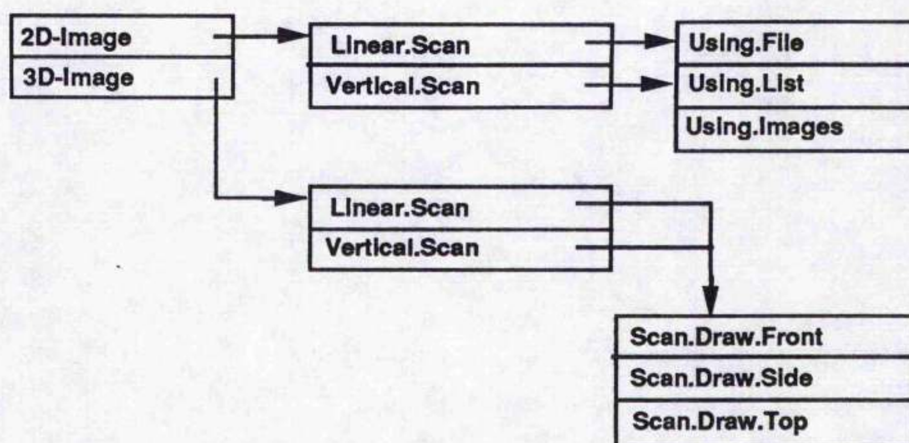
Most graphics packages make use of menus. Menu design helps in relieving the burden of remembering input options. All the options are presented in the form of a menu. Menus can easily be changed to accommodate different applications. They can be used as an input tool for operations and parameter values. The selection for the operations to be performed can be made by positioning a cursor at a menu position. Each menu option can have submenus. A selection from the first menu brings up a menu at the second level and so on. Each menu option is connected with the task (or algorithm) to perform. When we select an option, if it does not have submenus then the related programme will be performed asking for the input parameters if any. The menu design which we made is shown below. Each operation has either submenus or not.

Menu
Murray.Scans
Scan+Draw
Scan.Conversion
Scaling
Connectivity
Set.Operations
Superimposition
Hidden.Surface.Rm
Shading

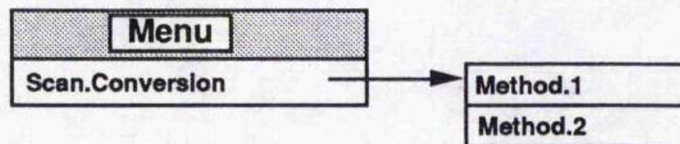
Now we consider each option separately and will discuss the proposal behind it. All the options are discussed in detail in the previous chapters. The first option with its submenus is shown below,



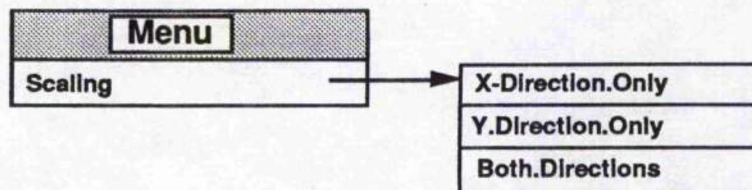
This option draws murray curves with different radices. The option *Murray.Scans* has two sub-options. The first option *Cartesian* has three sub-options, where the first two options draw general linear and general vertical murray curves and the last one draws the mixed curve i.e., vertical as well as linear tiles, as explained in chapter 2. The second option *Polar* draws polar murray scans. It has two options; one for the linear case and the other one for the vertical case. The shape of the curve depends upon the input radices. Each option will ask the user to input the radices.



The second option *Scan+Draw* takes an image as an input and then scans it to get the collection of runlengths, which is later on used to draw the image anywhere on the screen. This option can also take a file or a list as an input, this file is containing the collection of runlengths obtained after scanning the image. The option *Scan+Draw* has two sub-options. The first sub-option takes a 2-dimensional image whereas the second option considers a 3-dimensional image. The option *3D.Image* scans the images from the front side, left side and top side. The scanning pattern can either be vertical or linear depending upon the radices.

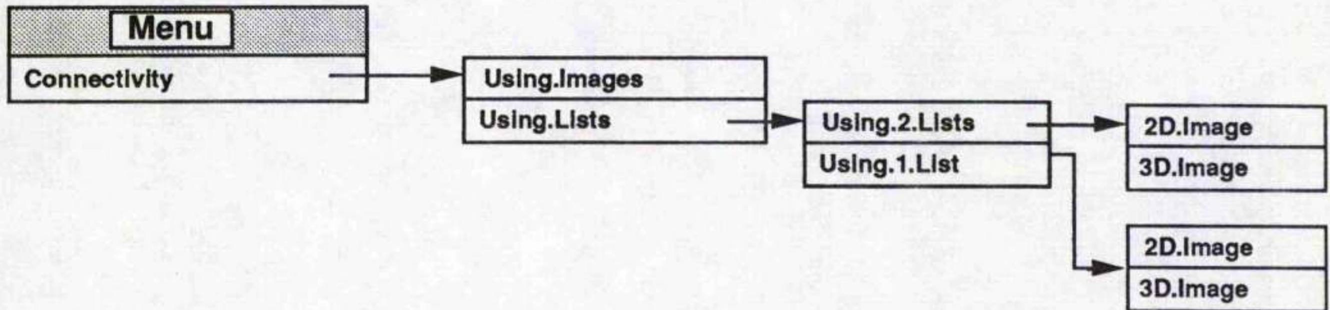


The option *Scan.conversion* (shown above) converts a horizontal scan into the vertical scan or vice-versa. It has two options. The first option uses murray arithmetic to convert one scan into another and the second option uses simple mathematical calculations to do the job. In both cases the input is murray runlengths and the murray radices. Both the methods are discussed in detail in chapter 4.

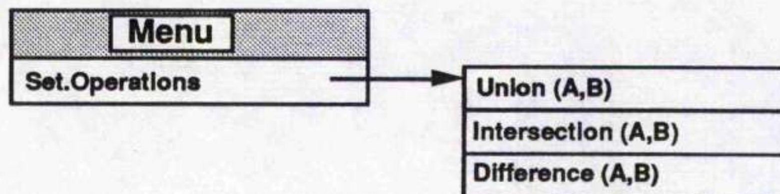


This option i.e., *Scaling,* scales the images up and down as required. It has three sub-options, the first option scales the images in the x-direction, the

second one scales the images in the y-direction and the last sub-option scales the images in both directions i.e., x as well as y-direction. More detail can be obtained from chapter 4.

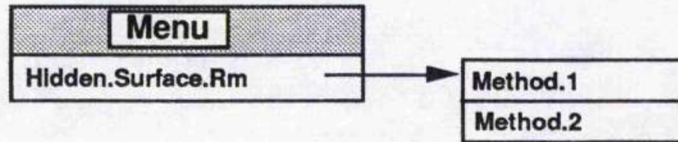


The option *Connectivity,* helps in extracting out a big chunk or a required area from a given image. It has two sub-options. In the first case it takes an image as an input and finds the required connected area. In the second case it takes either two lists of runlengths or one list of runlengths, which are obtained after scanning the images, to identify the homogeneous connected area. Refer chapter 6.

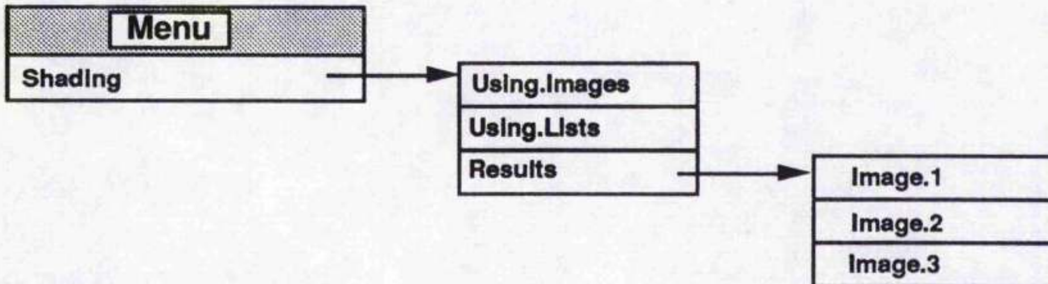


The option *Set.Operations,* merges two or more images together to give a single image. The merging of two or more images can be done to obtain the total black area in the images or the area which is common in the images or the area which is not common in the images. The input can either be the collection of images which has to be merged or the runlengths corresponding to the images which we want to merge (see chapter 5).

The next option *Superimposition,* superimposes one image on top of the other. It has no sub-options. The input can either be the two images or the runlengths corresponding to these images. See chapter 5.



The option *Hidden.Surface.Rm,* has two sub-options. Both the options remove the surfaces which are hidden from a given view point. The first sub-option scans all the planes using a suitable murray scan and then merges them together, and the second sub-option scans only the first plane and then merges the other planes one by one.



This option *Shading,* shades the visible area which has been obtained after removing the hidden surface. It takes a 3D.image or a list of runlengths which has been obtained after removing the hidden surface as an input. In the first case it will remove the hidden.surface and then shade the visible surface. The option *Results* contains the result obtained after using the above algorithms on the different images.The complete menu format is shown in Figure 8.1. The time obtained in processing different options which are coded in PS-algol and C is shown in Table 8.1.

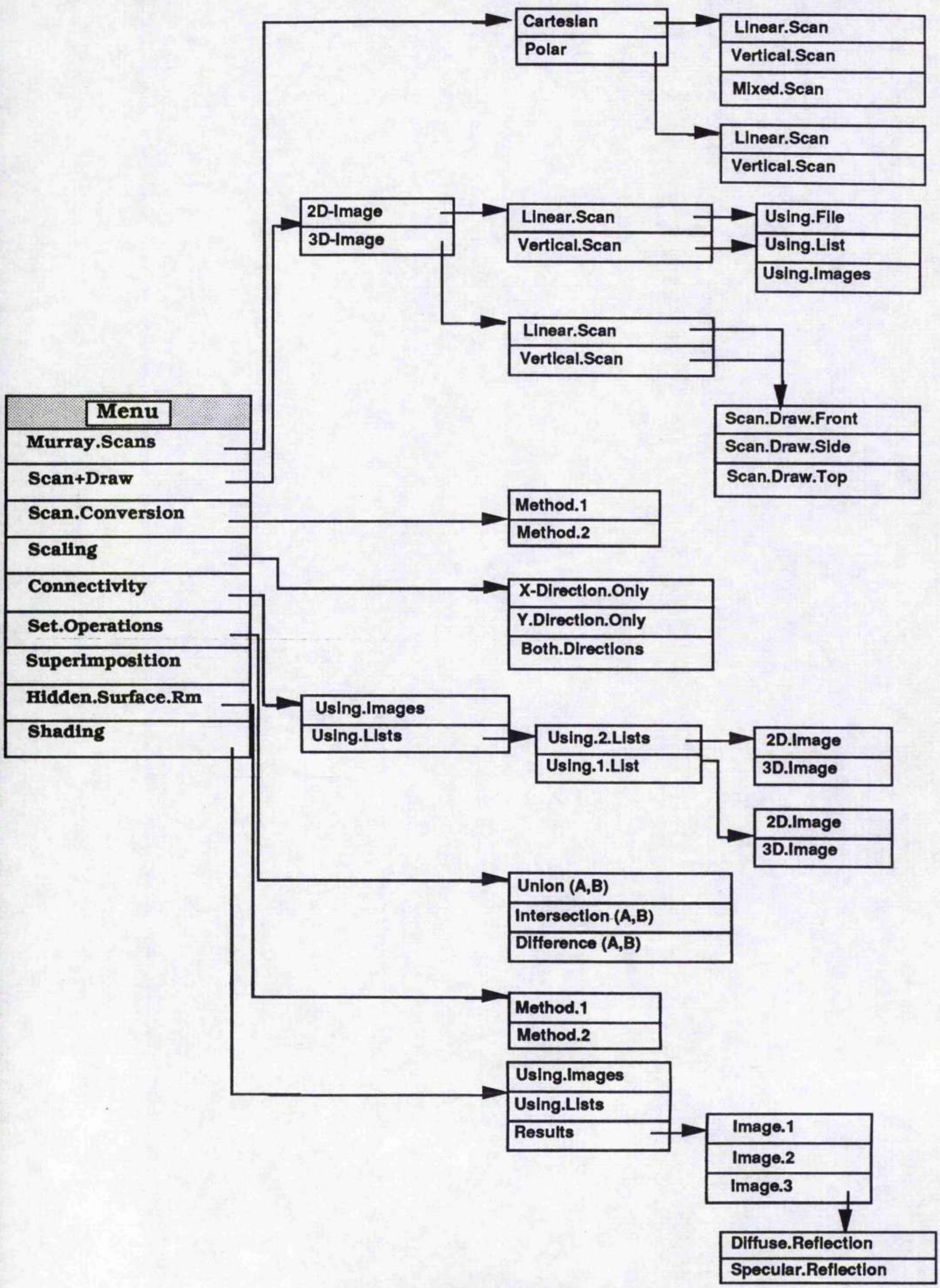


Figure 8.1. Menu Format

Options	Size of an image		Time (in secs)	
			PS-algol	C
Murray Scan	99*99 256*256		1.50 secs 9.05 secs	0.05 secs 0.2 secs
Polar Murray Scan	99*99 256*256		3.05 secs 14.52 secs	1.45 secs 4.20 secs
	Initial	Final		
Scaling	99*99 117*117	153*189 189*189	2.40 secs 3.02 secs	0.37 secs 0.42 secs
	No. of Planes	Size		
Intersection	2 2	99*99 99*99	3.44 secs 5.45 secs	0.5 secs 0.13 secs

Table 8.1
Time comparison between PS-algol and C.

Chapter 9

9. CONCLUSIONS AND FUTURE WORK 252

9 - 1 Conclusions 252

9 - 2 Future Work 256

9-1 Conclusions :

This thesis investigates the use of murray polygons to process arbitrary images e.g., satellite images, medical images, or any other image obtained from some device e.g., a vision capture system usually, *a camera.* Many associated problems related to image processing are solved by using the murray approach and are compared with those already defined for linear or quadtree(or octree) approaches. Initially all the methods are compared with each other and finally the results obtained by these methods are compared.

Standard Linear scan)**Vs** (Murray Scan or Quadtree or Octree Encoding :

Since in the case of the standard linear scan with flyback we require fewer calculations than that of the murray and quadtree or octree approach, hence in most of the cases the standard linear scan will be faster than the other two. But the standard linear approach does not, in general, give better exact compression than that of the murray approach or the quadtree or octree approaches. The reason is, it is one dimensional in nature. The coherence between the pixels is exploited only in left or right directions. Further each flyback results in the break of runlengths thus giving more codes (or runlengths) than the murray approach and quadtree or octree approaches.

As remarked earlier a standard linear scan with fly-back will take less time to encode an image than that of the murray approach, however hardware can be built to compensate for this. Similarly for the quadtree or octree approaches also.

Murray Approach) **Vs** (Quadtree or Octree Approaches :

In comparison to quadtree or octree approaches the murray approach may take less time to encode an image. The reason is the extra preprocessing steps used in forming the quadtree or octree. The preprocessing steps include,

1. conversion from rasters(i.e., runlengths) into quadtree;
2. merging groups of four pixels or blocks of a uniform color.

The scanning is generally done by a standard linear scan with flyback. In the case of the linear quadtree or octree, condensation and sorting has to apply to the collection of codes obtained after transforming rasters into a quadtree. In the case of the murray approach the only step to do is to encode an image using a murray scan, ignoring all other preprocessing steps. No processing has to be done once the codes have been obtained.

For obtaining exact compacted codes, the murray approach and quadtree or octree approaches will be equally effective, depending upon the shape of the images. In some cases the quadtree or octree approaches will be better and in some cases the murray approach will be better.The linear quadtree approach where only black pixels are stored, may be more compressive in some of the cases than that of the murray approach. But in the case of the linear quadtree approach the smaller the black homogeneous quadrant the bigger will be the code length and in the case of the murray approach the smaller the black area the smaller the code length, and vice versa. Further a murray polygon may have a better chance of capturing more pixels of the same color than that of the quadtree methods. The reason is, in the case of the quadtree or octree approaches each quadrants/octants are dealt separately to encode the information, whereas in the case of the murray approach we deal with the whole image. From the above facts we cannot say positively about the two approaches, as to which one is better. The best/worst cases are very much dependent upon the contents of the image.

The main advantage of the murray approach over quadtree or octree approaches is the size of an image to be processed. The murray approach can process any arbitrary rectangular image with no restrictions on the size, whereas a quadtree or octree approach can only directly process an arbitrary image if it is a square /cube of side length 2^n.

In the next few paragraphs the results obtained using the murray approach are surveyed. All the results are compared with those of the quadtree or octree approaches. We do not use the standard linear approach for comparison since in general it does not give better exact compression although it may be very fast for some of the operations.

Results :

In the case of scaling the images, it has been noted that the murray approach is slightly better than that of quadtree or octree approaches. The reason for this is the flexibility in the size of an image. In the case of the quadtree approach the scaling factor can be 2^n, whereas in the case of the murray approach the scaling factor can be m/n, where the variables m and n are integers. The only restriction which we have in the case of the murray approach is with the x-factor(y-factor); the quotient for the x-factor(y-factor) should not be even. (*Note : The x-factor(y-factor) depends upon the scan used to encode an image. If we are using a horizontal murray scan then the restriction will be with the x-factor otherwise with the y-factor).* But in the case of the murray approach we have to apply scan conversion algorithms, if we have to scale the images in both the directions, whereas in the case of the quadtree or octree approaches we do not require to change the codes. But since the murray approach is independent of the size of an image hence it may be more efficient in scaling up and down the images.

In the case of connected component labelling, both the approaches may take the same time to process, depending upon the shape of the image. But the quadtree or octree approaches may be more compact, especially the linear quadtree approach where only black pixels are stored. The reason is the use of linear murray scan which does not have the coherence between the pixels. The algorithm which uses a general murray scan may give better exact

compression in comparison to one with linear murray scan, but it will take more time to process an image. Further the exact compression can also be obtained, which will be comparable to that of the quadtree approach if we redraw the image using the runlengths obtained by linear murray scan and then scanning it using a general murray scan.

For hidden-surface removal both the methods are assumed to be equally effective. Both can process the arbitrary images plane by plane, which is advantageous if the image size is very large and the computer memory is very low. Both uses the same algorithm for all the surfaces, no special considerations being given to the curved surfaces.

For getting shaded images the murray approach has a slight advantage over the quadtree approach. As discussed earlier the intensity at a point in an image depends on the angle between the unit normal vector and the vector L towards the light source. The vector L can easily be obtained since we have the coordinates corresponding to the light source and to the point in an image. But to find a normal at a point we need at least two more points, so that we can find a plane passing through these three points and hence the normal. In case of quadtree (*Note : this quadtree is obtained after transforming an octree into a quadtree)* encoding, these two points can be calculated either by using 4-point connectivity or by 8-point connectivity of a point (i,j). The corresponding z-value can then be obtained from the linked list. But in the case of a murray scan the connected points with x, y, and z-values can be obtained straight from the scan. We do not have to do extra work in finding out the z-values corresponding to the connected points. Only at the boundary do we have to perform calculations to get the z-value.

Both the approaches are assumed to be equally effective for superimposition, set operations, and smoothing on the images.

9-2 Future Work :

Future work is required on a hardware implementation to increase the performance of the murray method in comparison with existing methods and applications.

Some investigations are required to convert a linear murray scan to a general murray scan, and vice-versa. This will be advantageous in some specific areas of digital image handling.

Obtaining reflection, refraction, and shadows from the images are other applications, where the use of murray polygons should be investigated.

Further work, presently under investigation is to obtain reflection from the images.

References

Abel, D. J. and Smith J. L.(1983), *'A data structure and algorithm based on a linear key for a rectangle retrieval problem,'* Computer Vision Graphics and Image Processing 24,1,1-13.

Annedda, C. and Felician, L.(1988), *' P-compressed quadtrees for image storing,'* The Computer Journal, 31, 4, 353-357.

Appel, A.(1968), *'Some techniques for shading machine rendering of solids,'* AFIPS, Spring Joint Comput. Conf., 37-45.

Atherton, P. R., Weiler, K. and Greenberg, D.(1978), *'Polygon shadow generation,'* Compter Graphics,12, 275-281.

Becmann, P. and Spizzichino, A.(1963), *'Scattering of electromagnetic waves from rough surfaces,'* MacMillan, N.Y.,1-33,70-98.

Bentley, J. L. (1975), *'Multidimensional binary search trees used for associative searching,'* Comm. ACM 18, 9, 509-517.

Bentley, J. L. and Stanat, D. F.(1975), *'Analysis of range searches in quadtrees,'* Inf. Process. Lett., 3, 6,170-173.

Blinn, J. F. and Newell, M. E.(1976), *'Texture and reflection in computer generated images,'* Comm. ACM,19, 542-547.

Blinn, J. F.(1977), *'Models of light reflection for computer synthesized pictures,'* Proc. SIGGRAPH, San Jose, Calif., 192-198.

Blinn, J. F.(1978), *'A scan line algorithm for the computer display of parametrically defined surfaces,'* Computer Graphics, Vol 12.

Blinn, J. F.(1978), *'Simulation of wrinkled surfaces,'* Computer Graphics,12, 286-292.

Bouknight, W. J.(1970), *'A procedure for generation of three dimensional half-toned computer graphic representations,'* Comm. ACM, 13, 9, 527.

Brown, R. C. (1955), *'Light,'* Longmans, Green and Co. London.

Buntin, I. M.(1988), *'The application of murray polygons to the compression of digital image data,'* Master's Thesis, University of St. Andrews.

Burton, F. W., Kollias, V.J. and Kollias, J.G.(1987), *'A general Pascal programme for map overlay of quadtrees and related problems,'* The Computer Journal, 30, 4, 355-361.

Carrick, R., Cole, A. J. and Morrison, R. (1987), *'An Introduction to PS-algol Programming(3rd edition),'* Persistent Programming Research Report, PPRR-31.

Catmull, E.(1974a), *'A subdivision algorithm for computer display of curved surfaces,'* Ph.D. Thesis, University of Utah.

Catmull, E.(1974b), *'Computer display of curved surfaces,'* Proc. IEEE Conf. Comput. Gr. Pattern Recognition Data Structure , 11.

Chien, C. H. and Aggarwal, J. K.(1986), *'Volume/Surface octrees for the representation of three-dimensional objects,'* Computer Vision, Graphics, And Image Processing, 36,100-113.

Cohen, Elaine,lyche,Tom and Riesenfeld, R. F.(1980), *'Discrete B-splines and subdivision techniques in computer aided geometric design and computer graphics,'* Compter Graphics, And Image Processing,14, 87-111.

Cole, A. J. (1966), *'Cyclic Progressive Number Systems.,'* Math. Gazette, vol. L, no. 372, pp. 122-131.

Cole, A. J. and Davie, A. J. T.(1969), *'Local smoothing by polynomials in n-dimensions,'* The Computer Journal, 12, 1, 35-41.

Cole, A. J. (1983), *'A Note on Space Filling Curves,'* Software-Practice and Experience, vol. 13, pp.1181-1189.

Cole, A. J. (1985a), *'A Note on Peano Polygons and Gray Codes,'* Intern. J. Computer Math, vol 18, pp. 3-13.

Cole, A. J. (1985b), *'Multiple Radix Arithmetic and Computer Graphics,'* Bulletin Inst. of Math. and its Applications, vol. 22, May/June, pp. 71-75.

Cole, A. J.(1985c), *'Direct Transformations between Sets of Integers and Hilbert Polygons,'* Intern. J. Computer Math., vol. 20, pp.115-122.

Cole, A. J. (1985d), *'Compaction Techniques for Raster Scan Graphics using Space-filling Curves,'* The Computer Journal, vol. 30, no. 1, pp. 87-92.

Cole, A. J. (1985e), *'Pure mathematics applied,'* Internal publication, CS/85/5, University of St. Andrews.

Cole, A. J.(1987), *'Data compaction using murray polygons,'* Computer Graphic Technology & Systems conference, CG87, London, pp. 185-194.

Cole, A. J. (1988a), *'Direct transformations for a class of space filling curves,'* Internal publication, CS/88/1, University of St. Andrews.

Cole, A. J. (1988b), *'Murray Polygons as a Tool in Raster Scan Graphics,'* ICONCG '88, Singapore, Sept. 88.

Cook, Robert, L.(1982), *'A reflection model for realistic image synthesis,'* Master's Thesis, Cornell University.

Cook, Robert, L. and Torrance, K. E.(1982), *'A reflectance model for computer graphics,'* ACM trans. on Gr. 1, 7-24.

Crow, F. C.(1977), *'The aliasing problem in computer generated shaded images,'* Comm. ACM 20, 11, 799-805.

Fine, H. B. and Thompson, H. D.(1909), *'Coordinate geometry,'* New York, The Macmillan Company.

Finkel, R. A. and Bentley, J. L. (1974), *'Quadtree: A data structure for retrieval on composite keys,'* Acta Inf. 4, 1, 1-9.

Foley, J. D. and Van Dam, A.(1982), *'Fundamentals of Interactive Computer Graphics,'* Reading, Mass. : Addison-Wesley Publishing Company.

Gardner, M. (!967), *'Mathematical games,'* Scientific American, March.

Gargantini, I. (1982), *'An effective way to represent quadtrees,'* Comm. ACM 25,12, 905-910.

Gargantini, I. (1983), *'Linear octrees for fast processing of three dimensional objects,'* Computer Graphics, And Image Processing,20, 4, 365 -374.

Gargantini, I., Walsh, T. R. and Wu, O. L.(1983), *'Viewing transformations of voxel-based objects via linear octrees,'* IEEE CG&A.. 12-21.

Gilbert, E.N.(1958), *'Gray codes and path on the n-cube,'* Bell System Technical J., 37,1, 815-826.

Goldschlager, L. M. (1981), *'Short algorithms for space filling curve,'* Software-Practice and Experience, 1, 403-410.

Gonzalez, R. C. and Wintz, P.(1987), *'Digital Image Processing,'* Addison-Wesley Publications.

Gouraud, H.(1971), *'Continuous shading of curved surfaces,'* IEEE Trans. On Computer 20, 6, 623 -629.

Gray, F.(1953), *'Us patent 2632058.'*

Griffiths, J. G. (1984), *'A depth coherence scanline algorithm for displaying curved surfaces,'* Comp. Aided Design 13, 2, 91-101.

Griffiths, J. G. (1985), *'Table-driven algorithms for generating space-filling curves,'* Comp. Aided Design,17, no.1, 37-41.

Griffiths, J. G. (1986), *'An Algorithm for Displaying a Class of Space-Filling Curves,'* Software-Practice and Experience, vol. 16(5), pp. 403-411.

Hearn, D. and Baker., M. P. (1986), *'Computer Graphics,'* USA : Prentice-Hall International.

Hilbert, D. (1891), *'Ueber stetige Abbildung einer Linie auf ein Flachenstuck,'* Math. Annln. vol. 38, pp. 459 - 460.

Hummel, J. A.(1965), *'Vectors,'* Addision - Wesley Publishing Co. Inc.

Hunter, G. M.(1978), *'Efficient computation and data structures for graphics,'* Ph.D. Dissertation, Department of Electrical Engineering and Computer Science, Princeton University, Princeton, N. J.

Hunter, G. M. and Steiglitz, K. (1979a), *'Operations on images using quadtrees,'* IEEE Trans. on Pattern Analysis and Machine Intell., PAMI-1(2), pp.145-153.

Hunter, G. M. and Steiglitz, K. [1979b]. *' Linear transformation of pictures represented by quadtrees,'* Computer Graphics, And Image Processing,10, 3,189-196.

Jaclins, C. and Tanimoto, S. L.(1980), *'Oct-trees and their use in representing three dimensional objects,'* Computer Graphics, And Image Processing,14, 3, 240-270.

Jones, L. and Iyenger, S. S.(1984), *'Space and time efficient virtual quadtree,'* IEEE Trans. on Pattern Analysis and Machine Intell., 6, 2, 244-247.

Kawaguchi, E. and Endo, T.(1980), *'On a method of binary picture representation and its applications to data compression,'* IEEE Trans. on Pattern Analysis and Machine Intell., 2, 1, 27-35.

Kelley, A. and Pohl, Ira.(1984), *'A book on C,'* The Benjamin/Cummings Publishing Company Inc. California.

Kennedy, H. C.(1973), *'Selected works of Giuseppe Peano,'* London, George Allen and Unwin Ltd.

Kernighan, B. W. and Ritchie, D. M.(1978), *'The C programming language,'* Prentice-Hall, Inc., Englewood Cliffs, New Jersey. 07632

Klerer and Korn (1967), *'Digital computer user's handbook,'* New York: McGraw- Hill.

Klinger, A. and Dyer, C. R. (1976), *'Experiments in Picture Representation using Regular Decomposition,'* Computer Graphics and Image Processing, no.5, pp. 68-105.

Lanczos, C.(1957), *'Applied Analysis,'* London, Sir Isaac Pitman and Sons, Ltd.

Mandelbrot, B. B. (1977), *'The Fractal Geometry of Nature,'* Publisher : W.H.Freeman and Company.

Meagher, D.(1982), *'Geometric modeling using octree encoding,'* Computer Graphics, And Image Processing 19, 2, 129-147.

Moore, E. H.(1900), *'On Certain Crinkly Curves,'* Trans. Am. Maths. Soc., vol 1, pp. 72-90.

Morrison, R.(1988), *'PS-algol reference manual,'* Fourth Edition, Persistent Programming Research Report 12.

Netravali, A. N. and Haskell, B. G.(1988), *'Digital pictures representation and compression,'* Plenum Press, New York.

Newell, M. E., Newell, R.G. and Sancha, T. L.(1972b), *'A new approach to the shaded picture problem,'* Proc. ACM, Natl. Conf., 443-450.

Newman, W. M. and Sproull, R. F., (1979), *'Principles of Interactive Computer Graphics,'* 2nd ed., McGraw-Hill, New York.

Null, A. (1971), *'Space filling curves, or how to waste time with a plotter,'* Software-Practice and Experience, vol 1, pp. 403-410.

Oliver, M. A. and Wiseman, N. E. (1983a), *'Operations on Quadtree Encoded Images.'* The Computer Journal, vol. 26, no. 1, pp. 83-91.

Oliver, M. A. and Wiseman, N. E. (1983b), *'Operations on Quadtree Leaves and Related Image Areas,'* The Computer Journal, vol. 26, no. 4, pp. 375-380.

Peano, G. (1890), *'Sur une courbe, qui remplit toute une aire plaine,'* Math. Annln., vol 36, pp. 157-160.

Phong, Bui-Tuong.(1975), *'Illumination for computer generated images,'* Comm. ACM 18, 6, 311-317.

Plastock, R.A. and Kalley, G. (1987), *'Theory and problems of computer graphics,'* McGraw-Hill, New York.

Raman, V. and Iyenger, S. S. (1983), *'Properties and applications of forests quadtrees for pictorial data representation,'* BIT 23, 4, 472-486.

Reddy, D. R. and Rubin, S.(1978), *'Representation of three dimensional objects,'* CMU-CS-78-113, Computer Science Dept., Carnegie-Mellon University, Pittsburgh.

Roberts, L. G. (1963), *'Machine perception of three dimensional solids,'* MIT Lincoln Laboratory, TR 315.

Rogers, D. F.(1985), *'Procedural Elements For Computer Graphics,'* New York: McGraw-Hill.

Rosenfeld, A. and Pfaltz, J. L.(1966), *'Sequential operations in digital image processing,'* 2nd ed. Academic Press, New York.

Rosenfeld, A. and Kak, A. C.(1976), *'Digital picture processing,'* Academic Press, New York.

Samet, H.(1981a), *'Connected component labeling using quadtrees,'* J. ACM 28, 3, 487-501.

Samet, H. (1981b), *'An algorithm for converting rasters to quadtrees,'* IEEE Trans. on Pattern Analysis and Machine Intell., 3, 1, 93-95.

Samet, H.(1982), *'Neighbor finding techniques for images represented by quadtrees,'* Computer Graphics, And Image Processing,18, 1, 37-57.

Samet, H. and Tamminen, M. (1984), *'Efficient image component labeling,'* TR-1420, Computer Science Dept., University of Maryland, College Park.

Samet, H. (1984), *'The quadtree and related hierarchical data structure,'* Computing Surveys, 16, 2, 187-260.

Samet, H. (1985), *'Data Structures for Quadtree Approximation and Compression,'* Comm. ACM, vol. 28, no. 9, pp. 973-993.

Schumacker, R.A., Brand, B., Gilliland and Sharp, W.(1969), *'Study for applying computer generated images to visual simulation,'* AFHRL, TR-69-14, US. Air force, Human Resources laboratory.

Scott, K. D.(1979), *'Transparency refraction and ray tracing for computer synthesized images,'* Master's Thesis, Cornell University.

Shneier, M.(1981), *'Calculations of geometric properties using quadtrees,'* Computer Graphics, And Image Processing, 16, 3, 296-302.

Sierpinski, W. (1912), *'Sur une nouvelle courbe qui remplit toute une aire plaine,'* Bull. Acad. Sic. Cracovie, Serie A, pp. 462-478.

Srihari, N. S.(1981), *'Representation of three dimensional digital images,'* Computing Surveys, 13, 4, 399-424.

Stevens, R., et al.(1980), *'Data ordering and compression of multispectral images using the Peano scan,'* IEE International Conference of Electronic Image Processing, no. 214.

Sutherland, I.E., Sproull, R. F. and Schumacker, R. A.(1974), *'A characterization of ten hidden surface algorithms,'* Computing Surveys, 16, 1-55.

Torrance, K. E. and Sparrow, E. M.(1967), *'Theory for off-specular reflection from roughened surfaces,'* Journal of the Optical Society of America, 57, 1105-1114.

Unnikrishnan, A. and Venkatesh, Y. V. (1984), *'On the conversion of raster to linear quadtrees,'* Department of Electrical Engineering, Indian Institute of Science, Bangalore, India.

Unnikrishnan, A., Venkatesh, Y. V. and Priti Shankar.(1987), *'Connected component labelling using quadtree-A bottom-up approach',* The Computer Journal, 30, 2, 176-182.

Warnock, J. E.(1969), *'A hidden surface algorithm for computer generated halftone pictures,'* Computer Science Dept. University of Utah, TR 4-15.

Watkins, G.S.(1970), *'A real time visible surface algorithm,'* Computer Science Department, University of Utah, UTECH-CSC-70-101.

Weiler, K. and Atherton, P.(1977), *'Hidden surface removal using polygon area sorting,'* Computer Graphics, Vol II, pp 214-222.

Whitted, T.(1978), *'A scan line algorithm for computer display of curved surfaces,* Computer Graphics 12.

Williams, L.(1978), *'Casting curved shadows on curved surfaces,'* Compt. Gr. 12, 270-274.

Wirth, N.(1976), *'Algorithms + Data Structures = Programs ,'* Prentice-Hall.

Witten, I. H. and Wyvill, B. (1983), *'On the generation and use of space-filling curves,'* Software - Practice and Experience, 6, 519-525.

Woodwark, J. R. (1982), *'The explicit quad tree as a structure for computer graphics,'* The Computer Journal 25(2), pp. 235-238.

Woodwark, J. R. (1984), *'Compressed Quad Trees,'* The Computer Journal, vol. 27, no. 3, pp. 225-229.

Wylie, C., Romney, G. W., Evans, D. C. and Erdahl, A.(1967), *'Halftone perspective drawings by computer,'* Proc. AFIPSF. JCC, 31-49.

Xiaoyang, M., Tosiyasu, L. K., Fujishiro, I. and Tsukasa, N.(1987), *'Hierarchical representation of 2D/3D gray scale images and their 2D/3D two way conversion,'* IEEE, Dec.

Yau, Mann-May, And Srihari, S. N.(1983), *'A hierarchical data structure for multidimensional digital images,'* Comm. ACM, 26, 7, 504-515.