ORIENTATION ADAPTIVE QUADTREES

by

Sri Hartati

TR90-053, September 1990

This is an unaltered version of the author's
M.Sc.(C.S.) Thesis

Faculty of Computer Science
University of New Brunswick
P.O. Box 4400
Fredericton, N.B. E3B 5A3

Phone: (506) 453-4566
Fax: (506) 453-3566

Orientation Adaptive Quadtrees

by

Sri Hartati

BSc — Gadjah Mada University

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Science

in the

Faculty of Computer Science

This thesis is accepted.

____

Dean of Graduate Studies and Research

THE UNIVERSITY OF NEW BRUNSWICK

July, 1990

© Sri Hartati, 1990

Abstract

This thesis addresses the area of spatial data structures. There are numerous hierarchical data structuring techniques in use for representing raster data. One commonly used method that is based on recursive decomposition is the quadtree. Its development has been motivated to a large extent by a desire to reduce the amount of space required to store raster data representations of objects.

The Orientation Adaptive Quadtree (OAQ) representation is a new quadtree representation which positions the quadtree axes centred at the object's centroid and aligned with the object's principal axis of inertia. This approach is explained along with a quick method for computing the orientation of the object.

Boolean operations on two OAQs representing two objects which differ in position, orientation, and resolution are also presented. The method maintains the quadtree structure, and it consists of two major operations : extracting polygons produced by intersecting every two blocks representing two specified nodes from different OAQs, and merging those polygons which results in an object to be represented as another OAQ. The result of Boolean OAQ opereation has its own orientation. OAQ representations for 17 different geographical objects are compared to their standard quadtree representations, with an average reduction of 9.5% in the number of nodes required for their representation. Two objects were also used to test the Boolean intersection and union operations on OAQs. OAQ representation of rectangular objects gives a reduction in the number of nodes of more than 80%.

Contents

Abstract ii

List of Tables v

List of Figures vi

Acknowledgements x

1 Introduction 1

1.1 Background 1

1.2 Quadtree representation 2

1.3 Quadtree properties 3

1.4 Purpose of the thesis 6

1.5 Outline of the thesis 7

2 OAQs 8

2.1 Determining the orientation of planar object 8

2.2 Coordinate transformation 18

2.3 Representing a tightly closed boundary 23

2.4 Run Length Code (RLC) representation 27

2.5 Quadtree Encoding 31

3 Boolean operations with OAQs 41

3.1 Definition and notation 43
3.2 Process of extracting polygons 44
3.2.1 Recording a block of OAQ B in OAQ A 46
3.2.2 An arbitrarily oriented block intersection 50
3.2.3 Storing polygons in a black node of the reference OAQ 53
3.3 Merging operations 55
3.3.1 Method I 59
3.3.2 Method II 67
3.3.3 Updating polygons 68
3.4 Intersection operation between two OAQs 73
3.5 Union operation between two OAQs 80

4 Assessment **91**
4.1 Simulated test cases 91
4.2 Real test cases 93
4.3 Analysis 99
4.3.1 Analysis of intersection operation 106
4.4 Analysis of union operation 114

5 Conclusions **117**
5.1 Suggestions for futher work 118

References **120**

List of Tables

2.1 The results of computing object's orientation. 18

4.1 Number of nodes for Figure 4.1. 92

4.2 The execution time of Boolean OAQ operations of Figure 3.8 93

4.3 Encoding objects using OAQs and standard quadtrees. 98

List of Figures

1.1 Example of quadtree representation. 4

2.1 Example of standard quadtree encoding of an object not aligned to the quadtree axes. 9

2.2 Simple planar graph embedded in the plane. 12

2.3 Evaluation of double integral. 14

2.4 Planar graph embedded in the plane with $b = 0$. 16

2.5 Coordinate systems 1 (original) and 2 (object space). 21

2.6 Object space and image space. 24

2.7 A closed boundary example. 26

2.8 State diagram of an algorithm generating TCB. 27

2.9 The AUGMENT algorithm for augmenting the raster boundary of a closed polygon. 28

2.10 Example of RLC. 30

2.11 The compact representation of RLC. 30

2.12 An example of RLC compact representation. 30

2.13 The TO_RLC algorithm for generating a run-length coded (RLC) description of the graph. 31

2.14 Pascal record structure for an orientation adaptive quadtree. 32

2.15 Morton matrix. 33

2.16 An image, its maximal blocks and corresponding quadtree. 34

2.17 Subtree in the process of obtaining a quadtree corresponding to Figure 2.21(b). 35
2.18 An oriented image, its maximal blocks and the corresponding OAQ. . 36
2.19 Linear coding for the OAQ of Fig.2.19. 37
2.20 The QT_MAKE algorithm for controlling the final quadtree. 37
2.21 The IMAGE algorithm for determining if image pixel (x,y) is BLACK or WHITE. 38
2.22 The CONSTRUCT algorithm for constructing quadtree. 39
3.1 Intersection of two standard quadtrees. 42
3.2 The Pascal record structure for Boolean OAQ operations. 44
3.3 An example of the intersection of two OAQs. 47
3.4 Outer blocks of OAQ A and B. 48
3.5 Intersection of two black blocks. 51
3.6 Four possible cases of clipping block. 52
3.7 The ARBOR_BLOCK_INT algorithm for extracting a polygon produced by intersecting between two specified blocks. 54
3.8 Intersection of two OAQs. 56
3.9 The intersection of the block of OAQ A and the blocks of OAQ B. . . 57
3.10 A set of edges representing polygon R. 57
3.11 Steps of extracting polygons for Boolean operations with two OAQs. . 58
3.12 The MERGE_OPERATION algorithm for merging polygons for boolean OAQ operations. 60
3.13 The MERGE_OPERATION_B algorithm for merging all polygons which are stored in the node of OAQ A; i.e. BLACK node for intersection operation, WHITE node for union operation. 61
3.14 The MERGE_OPERATION_A algorithm for merging all polygons which are stored in GRAY children nodes. 62
3.15 Subtree of OAQ B with its polygons. 64

3.16 Possible combinations of polygon types being merged. 66
3.17 The temporary subtree of OAQ B. 68
3.18 The CREATE_SUBTREE algorithm for creating a temporary subtree of OAQ B. 69
3.19 The MERGE_SUBTREE algorithm for merging polygons which belong to the subtree of OAQ B. 70
3.20 Blocks representing a GRAY node. 71
3.21 The subtree of an OAQ illustrating polygon updating. 72
3.22 Process of updating polygon P and Q. 73
3.23 The UPDATE_POLYGON algorithm for updating a polygon to be a polygon owned by its parent node. 74
3.24 The Venn diagram of the intersection of A and B. 74
3.25 The INT_CHECK_NODES algorithm for examining nodes of OAQ A and B for intersection Boolean operation. 76
3.26 Intersection of two line segments. 77
3.27 A set of edges defining the result of intersection between two OAQs. . 81
3.28 The BOOL_OAQ_OPER algorithm for doing Boolean OAQ operations. 82
3.29 Union of A and B. 83
3.30 The complement of A. 83
3.31 The intersection of the complement of A and B. 84
3.32 The union of A and $(\overline{A} \cap B)$. 84
3.33 The subtree of union operation between two OAQs. 85
3.34 The UNION_CHECK_NODES algorithm for extracting polygon for union Boolean operation. 87
3.35 Union of two OAQs. 88
3.36 The result of Boolean union operation of two OAQs from Figure 3.32. 90
4.1 Example of an object encoded as OAQ, original object corner coordinates are (-0.5,5), (0.5,5), (0.5,-5), (-0.5,-5). 92

4.2 OAQ representation of the objects A and B with object to image scale factors of A = 2 and B = 1. 93
4.3 OAQ representation of the objects A and B with object to image scale factors of A = 1 and B = 0.5. 94
4.4 The result of the Boolean intersection operation for OAQs A and B of Fig.4.3. 96
4.5 The result of the Boolean union operation for OAQs A and B of Fig.4.3. 97
4.6 The original objects of a forest area and Tower lake. 100
4.7 The OAQ representations of the forest and the lake. 101
4.8 The result of the intersection OAQ operation of Figure.4.7, and its OAQ representation. 102
4.9 The result of the union OAQ operation of Figure.4.7, and its OAQ representation 103
4.10 An example of the intersection operation. 110
4.11 Example where the OAQ B lies inside OAQ A. 113

Acknowledgements

I wish to express my sincere thanks and appreciations to my supervisor Dr. Brad George Nickerson for his useful advice, guidance, and contructive criticism given throughout the extent of the research. Dr Nickerson initially identified OAQ as a quadtree representation when accounting for the orientation of the object. It might be used in spatial data processing.

I greatly appreciate and thank to the Goverment of Indonesia for the financial support and the opportunity to study together with my husband at the University of New Brunswick in Canada.

I would also like to express my deepest feeling of gratitude to in particular : my husband Agus Harjoko for his wonderful support and his understanding during the difficult hours of my work; my mother Ibu Wiryodaryanto and my parent in law Bp. Ibu Sosrowidarsono for their moral and other support during my study.

Chapter 1

Introduction

1.1 Background

Region representation is widely used in computer graphics, image processing and geographical information systems. Regions can be represented by converting their boundary to chain code representation [8], or shape number representation [2]. It can also be represented by the skeleton of the region using the medial axis transform (MAT) [3]. Those representations are compact, but they are not well suited for set theoretic operations. For binary region representation, there are several coding techniques that can compress the data even further. A popular technique is called run-length-encoding which is reviewed in section 2.4. This representation results in significant storage saving since only the length information needs to be stored. An alternative region representation, which provides a compact hierarchical representation, is the quadtree representation. It is based on a collection of maximal blocks that are in the given region. It is convenient for performing operations such as union and intersection, and facilitating search. In addition, it is also a useful technique when memory size limitations preclude storing in core a 2D array corresponding to the region.

1.2 Quadtree representation

The quadtree is a class of data structures used to represent spatial data in a plane. It can also represent a region from a raster representation. It is based on the principle of recursive decomposition of 2D space.

The motivation for development of the quadtree is to reduce the amount of space required to store raster data representations of objects. It is also intended to facilitate better performance in terms of execution speed for certain operations such as point location and area calculation. Algorithms using quadtrees have execution times that are dependent on the number of blocks in the object, and not on their size.

In many real-world applications the size of the quadtree representing the object is very large due to the orientation of a regularly shaped object. Space requirements of the quadtree representation can exceed the amount of space that is available. This is sometimes due simply to the relative orientation of the object and the coordinate frame used for the quadtree.

This research attempts to find the best representation for quadtrees when accounting for the orientation of the object. A method for quadtree data storage was developed which is sensitive to the orientation of the object. This resulted in a representation which is optimal in terms of storage requirements by accounting for large second order moments of objects.

The quadtree representation is based on a successive subdivision of an image into homogeneous blocks. If the image is all one color, then it is represented by a single block, otherwise the image is decomposed into quadrants, subquadrants,$\cdots$ until each block is homogeneous. The region can be represented by a 2D binary array. If the array does not consist entirely of 1's or 0's, then the image array is divided into four equal-sized quadrants, subquadrants $\cdots$ and so on, until blocks contain entirely 1's or 0's. If the block consists of 1's, it corresponds to the image inside the region and is said to be ' black ', otherwise it corresponds to the image outside of the region

and is said to be ' white '. This black or white block could be a single pixel. The relationship among quadrants are represented as a tree of out-degree four, with the root node representing the entire image array. The four sons of the root node represent the quadrants (labelled in order NW, NE, SW, SE) and the leaf nodes correspond to those blocks of the array for which no further subdivision is needed. The non-leaf nodes correspond to blocks which need further subdivision.

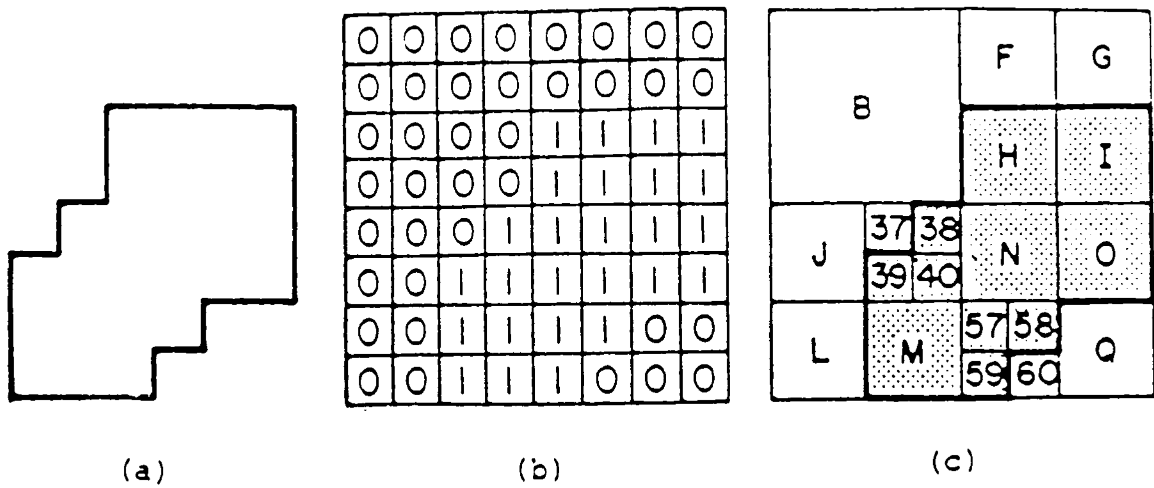
Each block has a standard size and position. For a 2^n by 2^n image, the root node is said to be at level n, and nodes in the tree represent blocks of size 2^{level} by 2^{level}. For example, Figure 1.1 (a) illustrates the region quadtree, which is represented by a $2^3 \times 2^3$ binary array image shown in Figure 1.1 (b). Consider that 1's correspond to image elements inside the region (pixel) and 0's correspond to image elements outside the region. The resulting quadtree block decomposition is shown in Figure 1.1 (c). Figure 1.1 (d) shows the quadtree structure.

Since the quadtree representation is based on a successive subdivision of an image into homogeneous blocks, it is a useful encoding of the spatial occupancy array, and it is effective for representing binary images. For non binary images, the quadtree representation may not result in significant storage saving.

1.3 Quadtree properties

The quadtree representation allows one to easily compute geometric properties such as perimeter [28], and area, and to perform searching operations such as point inclusion [30].

Performing operations on the quadtree representation is similar to most operations carried out in tree structures. In fact, most tree structures provide us with two basic steps for performing operations. The first step is either traversing the tree in a specified order or constructing the tree. The second one is performing a computation at each node by making use of its neighbouring nodes; that is, nodes representing



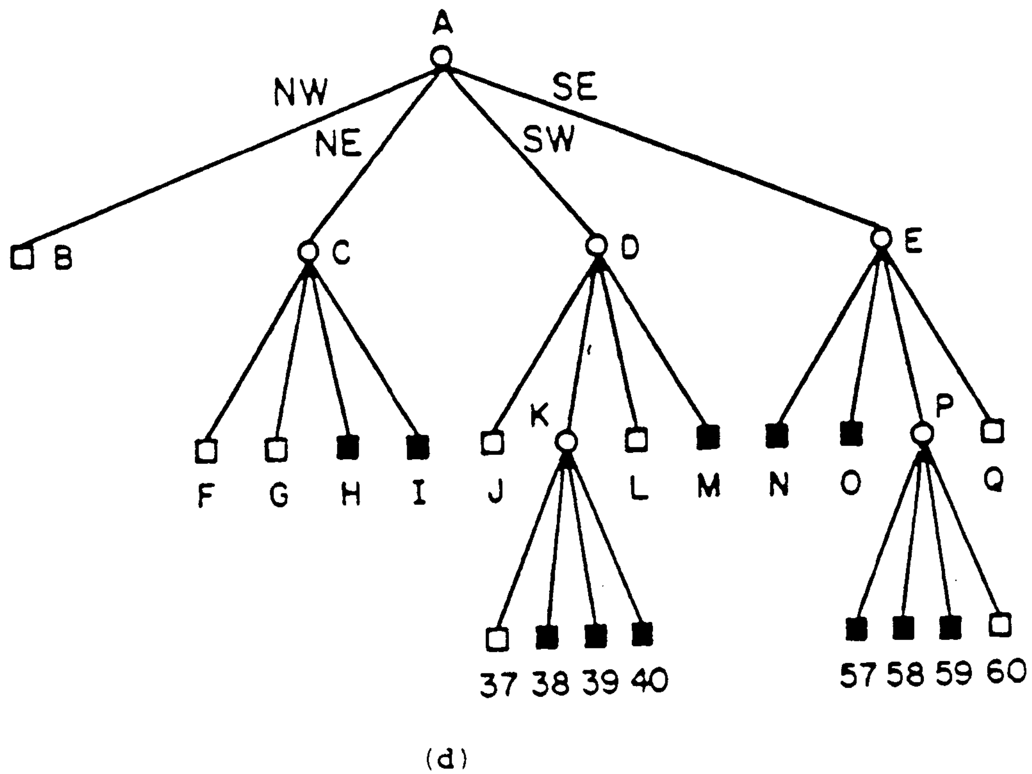


Figure 1.1: (a) Region. (b) Its binary array representation. (c) Its block decomposition. (d) Quadtree representation of the blocks (from [25]).

image blocks which are adjacent to the given node's block. Those two basic steps can be carried out successively or often they can be carried out in parallel. The following section shows us the operations carried out in the quadtree by making use of the quadtree properties.

To determine whether a point is inside a given region or not, a traversal from the root of the tree is required. We start from the root of the quadtree and use the x and y coordinate of the center of its node to determine which of the four subtrees contains the given point. The center of each node can be computed by knowing the level of the node. For example, given a 512×512 image, the level of the root is $n = 9$, the northeast-most coordinate is (512,512), and the level of the sons of the node is $n - 1 = 8$. The defining coordinates of the son's blocks can now be calculated. The NW son's upper right coordinate is (256,512), the NE son's is (512,512) the SW son 's is (256,256) and the SE son's is (512,216).

In general, if the traversal visits non-root nodes with level k, the coordinates of the node's children can be calculated. For the NW node the upper right coordinates are (2^k,2^{k+1}) ,for the NE node $(2^{k+1}, 2^{k+1})$, for the SW node $(2^k, 2^k)$ and for the SE node is $(2^{k+1}, 2^k)$. The centre coordinates of the nodes can also be calculated easily. Therefore the location of a point being tested for inclusion can be found easily. For example, if both x and y coordinates of a point being tested is less than the x and y coordinates of the center of the root node, then the point being tested is located in the SW of subtree of the root. This algorithm is applied recursively to the SW of the subtree of the root until it is determined that the point is inside a leaf node. If it is white, the point is outside the region, if black, the point is inside the region.

The area calculation can be performed by summing the area over the black leaf nodes. The area of each black leaf node can be calculated by knowing its level and performing calculation $2^n \times 2^n$, where n is the level of the black leaf node.

1.4 Purpose of the thesis

The work for the thesis is concentrated on developing a method for quadtree data storage which is sensitive to the orientation of an object. It is also intended to result in a reduction in the storage requirement for regular objects which are not aligned with the quadtree frame.

This work has several objectives, as follows :

1. To develop an orientation adaptive quadtree representation to store the raster representation of an object, along with algorithms for Boolean operations on these orientation adaptive quadtrees.

2. To compare the storage and execution efficiency of this approach with the standard quadtree method.

3. To investigate an orientation adaptive quadtree representation of linear features such as roads.

4. To perform an evaluation of this method using real 2D map data.

The approach to deal with the rotated image is based on the orientation of the object. To calculate the orientation of the object, it's second order moments were computed before converting the object to the Orientation Adaptive Quadtree (OAQ) representation.

Knowing the orientation of the object, the coordinate frame of the quadtree was aligned with the principle axis of the object. The orientation of the object and the centroid of the object become part of the OAQ data structure. The details for representing an object as an OAQ are explained in chapter 2.

1.5 Outline of the thesis

This thesis is organized in the following way: Chapter 1 provides an overview of the quadtree data structure. Chapter 2 details the Orientation Adaptive Quadtree(OAQ) representation including a complete method for computing the OAQ representation.

Set theoretic operations on OAQs are discussed in chapter 3. The algorithms are developed by maintaining quadtree properties. The details of these set operations are presented.

Experimental work and the results of these experiments are covered in chapter 4. This includes some simulated test cases and real test cases. The investigation of OAQ representation for linear features such as roads and the investigation of OAQ representation methods using real 2D map data for the Fredericton area are discussed. An analysis of the execution efficiency of OAQ algorithms is discussed as well.

Chapter 5 concludes all discussions and presents some recommendations for future work.

Chapter 2

OAQs

Objects which are not aligned to the object space axes, and which have a large eccentricity (see the example in Figure 2.1) may require many nodes for the standard quadtree encoding. If the quadtree axis was aligned with the principal axis inertia, considerable savings in the number of nodes required would result.

The OAQ representation is a quadtree representation which takes advantage of any second order moment of inertia of planar objects by centering the quadtree axes at the object's centroid and aligning the quadtree axes with the object's principal axis of inertia.

This representation provides a reduction in the number of leaf nodes necessary to store the quadtree compared to a quadtree aligned with the object coordinate axes. The following sections gives the details on converting a given planar graph into an OAQ.

2.1 Determining the orientation of planar object

OAQ representation of any planar object requires a calculation of its orientation. Several approaches to determining the orientation of a planar object exist. One such method uses the the Hotelling transform to compute the principal eigenvectors which

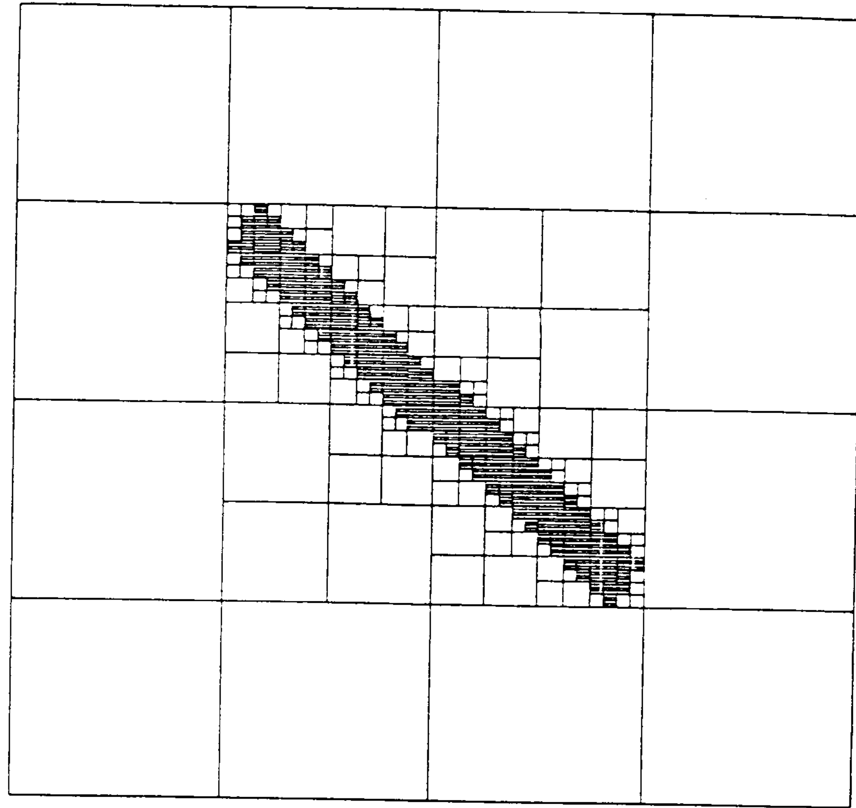


Figure 2.1: Example of standard quadtree encoding of an object not aligned to the quadtree axes.

can be used to determine principal axis orientation [5]. This method was not used here. Instead, the second order moments for a complete planar object represented by an embedded graph of even degree were derived in such a way that the computations are dependent on the number of edges. The second order moment was computed by adding the contribution of all the edges, taking into account whether or not the edge is horizontal or vertical. This section derives the equations used to compute the second order moments, and subsequently the orientation of the principal axis of inertia.

In general, the two dimensional $(p+q)$th order moments of some function $f(x,y)$ defined on the Cartesian plane are given in terms of the following integral [11]

$$m_{pq} = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} x^p \, y^q \, f(x,y) \, dx \, dy \tag{2.1}$$

for $p, q = 0, 1, 2, \cdots$. For the discrete case, space $f(x,y)$ represents a closed boundary region R, and the moments are computed as a sum over the area within the boundary with $f(x,y) = 1$.

The central moments are defined as

$$\mu_{pq} = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} (x - \overline{x})^p \, (y - \overline{y})^q \, f(x,y) \, dx \, dy \tag{2.2}$$

for $p, q = 0, 1, 2, \cdots$, and where $(\overline{x}, \overline{y})$ represent the coordinates of the centroid. In terms of moments, the centroid is computed as

$$\overline{x} = \frac{m_{10}}{m_{00}}, \qquad \overline{y} = \frac{m_{01}}{m_{00}} \tag{2.3}$$

where m_{00} is the area.

The central moments can also be expressed in terms of the ordinary moments [11, 1] as

$$\begin{aligned} \mu_{00} &= m_{00} = \mu \\ \mu_{10} &= \mu_{01} = 0 \\ \mu_{11} &= m_{11} - \mu \overline{x} \, \overline{y} \end{aligned} \tag{2.4}$$

$$\mu_{20} = m_{20} - \mu\overline{x}^2$$

$$\mu_{02} = m_{02} - \mu\overline{y}^2$$

The orientation, θ, of the principal axis of inertia is computed as [11,1]

$$tan2\theta = \frac{2\mu_{11}}{\mu_{20} - \mu_{02}} \tag{2.5}$$

In computer vision, the ij^{th} moments provide a measure of eccentricity or elongation [2]. For a digital image equation (2.2) is defined as

$$m_{pq} = \sum_{(x,y)\ in\ R} (x - \overline{x})^p (y - \overline{y})^q \tag{2.6}$$

where $(\overline{x},\overline{y})$ is the mean vector of the points inside R, defined as

$$\overline{x} = \frac{1}{k} \sum_{x\ in\ R} x, \quad \overline{y} = \frac{1}{k} \sum_{y\ in\ R} y \tag{2.7}$$

where k is the number of points in the entire region. The orientation, θ, is given in equation (2.5) and the approximate eccentricity e is defined as [2]

$$e = \frac{(\mu_{20} - \mu_{02})^2 + 4\mu_{11}}{area} \tag{2.8}$$

where $area$ is the area of the entire region. The second order moments can be calculated using equation (2.6). This computation does not give an accurate result for the orientation, θ, since the values of (x, y) are values of pixel coordinates.

Several approaches based on numerical integration such as Simpson's rule, or trapezoidal rule can be used to calculate the orientation, θ. For example, consider Figure 2.2. The second order moment m_{20} under line segment (x_0, y_0) - (x_1, y_1) is defined from equation (2.1) as

$$m_{20} = \int_{x_0}^{x_1} \int_{y_0}^{y_0+bx} x^2 y^0 \, dy \, dx \tag{2.9}$$

If $Q(x) = \int_0^{y_0+bx} f(x,y)\, dy$ then

$$m_{20} = \int_0^{x_1} Q(x)\, dx \tag{2.10}$$

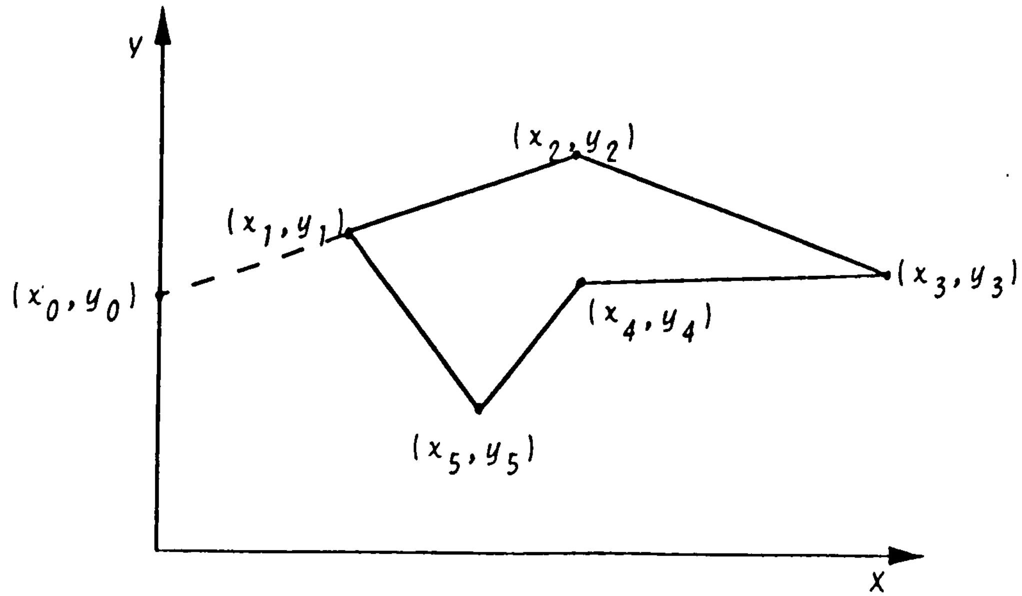


Figure 2.2: Simple planar graph embedded in the plane.

If Q(x) were known, any of the standard algorithms could be applied to the numerical determination of m_{20}, such as the trapezoidal rule or Simpson's rule. Consider the evaluation of the function of equation (2.10). Let $[x_0, x_1]$ be a finite interval on the x axis which has been partioned into j subinterval called *panels*, $[a_i, a_{i+1}], i = 1, \cdots, j$. Assume $a_1 = x_0$, and $a_{j+1} = x_1$, and $a_1 < a_2 < \cdots < a_{j+1}$. Let $h_i = a_{i+1} - ai$ be the panel widths. The evaluation of m_{20} can be computed using the trapezoidal rule $T(f)$ or $S(f)$ Simpson's rule [7] as

$$T(f) = \sum_{i=1}^{j} h_i \frac{f(a_i) + f(a_{i+1})}{2} \tag{2.11}$$

$$S(f) = \sum_{i=1}^{j} \frac{1}{6} h_i [f(a_i) + 4f(\frac{a_i + a_{i+1}}{2}) + f(a_{i+1})] \tag{2.12}$$

The calculations of second order moments for all specified line segments can be carried out using equations (2.10) and (2.12). These numerical approaches have the drawback of requiring a large amount of computations to obtain accurate results, since they are based on adding up the integrand at small intervals of the abscissa within the range of integration. The problem is to find the integral as accurately as possible with the

smallest number of function evaluations of the integrand. Several experiments were tried using numerical integration for calculating the orientation of a planar object. The result showed that a larger number of function evaluations of the integrand gives a higher accuracy, but this leads to longer execution.

Another method based on the classical formulation of integration was investigated. The second order moment can be evaluated directly from equation (2.1). Double integrals are required. Let first consider the evaluation of a definite integral

$$\int_a^b f(x)\,dx \tag{2.13}$$

The integrand is a function f(x) which exists for all x in an interval $a \leq x \leq b$ of the abscissa. In the case of a double integral, the integrand is a function $f(x,y)$ which is given for all (x,y) in a closed bounded region of the xy plane.

The double integral of $f(x,y)$ over the region R, is denoted as [13]

$$\int\int_R f(x,y)\,dx\,dy \tag{2.14}$$

This double integral may be evaluated by two successive integrations as follows. Consider Figure (2.3), and suppose that R can be described by inequalities of the form

$$a \leq x \leq b \qquad g(x) \leq y \leq h(x)$$

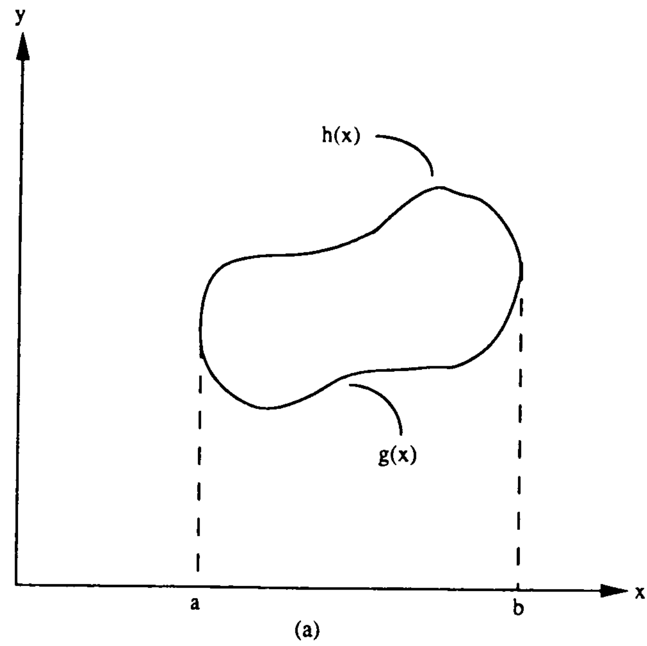
so that $y = g(x)$ and $y = h(x)$ represent the boundary of R at the bottom and top. Then

$$\int\int_R f(x,y)\,dy\,dx = \int_a^b \left[\int_{g(x)}^{h(x)} f(x,y)\,dy\right] dx \tag{2.15}$$

The integration of the inner integral in the equation (2.15) shows that x plays the role of a parameter, and the result of this evaluation will be a function of x, say $F(x)$. By integrating $F(x)$ over x from a to b we then obtain the value of the double integral.

In the case of R described by inequalities of the double form

$$c \leq y \leq d \qquad p(y) \leq x \leq q(y)$$



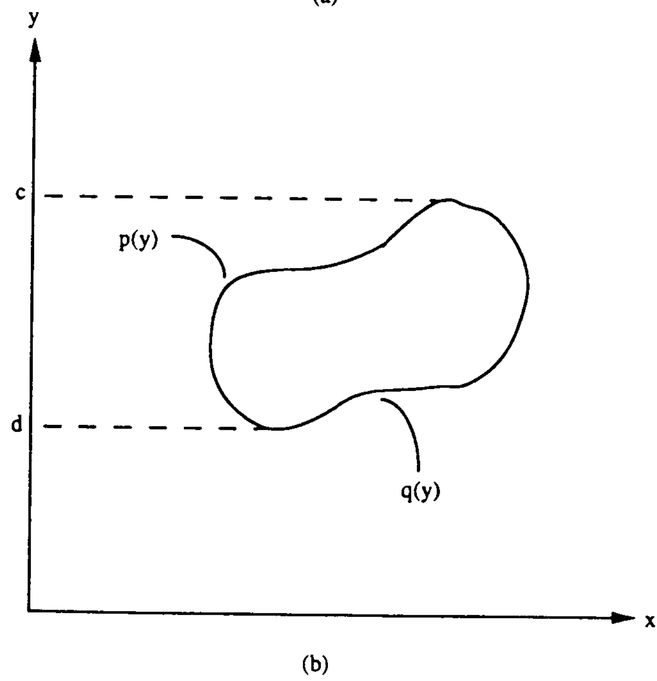


Figure 2.3: Evaluation of double integral.

the double integral 2.14 can be evaluated as

$$\int\int_R f(x,y)\,dy\,dx = \int_c^d \left[\int_{p(y)}^{q(y)} f(x,y)\,dx\right] dy \tag{2.16}$$

We first integrate the inner integral over x and then the resulting function of y is evaluated over y from c to d. This evaluation is depicted in Figure 2.3.

The region, R, may not be represented by these inequalities, but it can be divided into finitely many portions which have that property. To obtain the value of the double integral the integration of $f(x,y)$ is carried out over each portion separately, and the results are added together.

From the above considerations, the evaluations of the second order moment of planar graphs can be derived from equation (2.1). This double integral is evaluated over the region bounded by line segments. The derivation of the nth order moment with respect to the x axis is derived using equation (2.15), and the derivation of nth order moments with respect to y axis is done using equation (2.16). Consider the planar graph shown in Figure 2.2.

The second order moment m_{02} of the area under line segment $(x_0,y_0)-(x_1,y_1)$ can be defined as

$$m_{02} = \int_0^{x_1}\int_0^{y_0+bx} y^2\,dy\,dx \tag{2.17}$$

Evaluating the above equation in the manner describe by equation (2.15) results in

$$m_{02}^1 = \frac{1}{12b}(y_1^4 - y_0^4) \tag{2.18}$$

A similar evaluation can be carried out to compute the second moment under line segment $(x_0,y_0)-(x_2,y_2)$, which results in

$$m_{02}^2 = \frac{1}{12b}(y_2^4 - y_0^4) \tag{2.19}$$

The second order moment under line segment $(x_1,y_1)-(x_2,y_2)$ can be derived by taking the difference $m_{02}^2 - m_{02}^1$, which gives

$$m_{02}^{1,2} = \frac{1}{12b}(y_2^4 - y_1^4) \tag{2.20}$$

This equation shows that the second order moments of an area under a line segment of the boundary of a planar graph are dependent only on the slope of the line segment and end-point coordinates. In general, the second order moment for line segment $(x_i, y_i) - (x_{i+1}, y_{i+1})$ can be stated as

$$m_{02}^{i,i+1} = \frac{1}{12b}(y_{i+1}^4 - y_i^4) \tag{2.21}$$

If the slope b is zero or close to zero, as shown in Figure 2.4, an alternative formulation is used to avoid ill-conditioning. Evaluating the second order moment

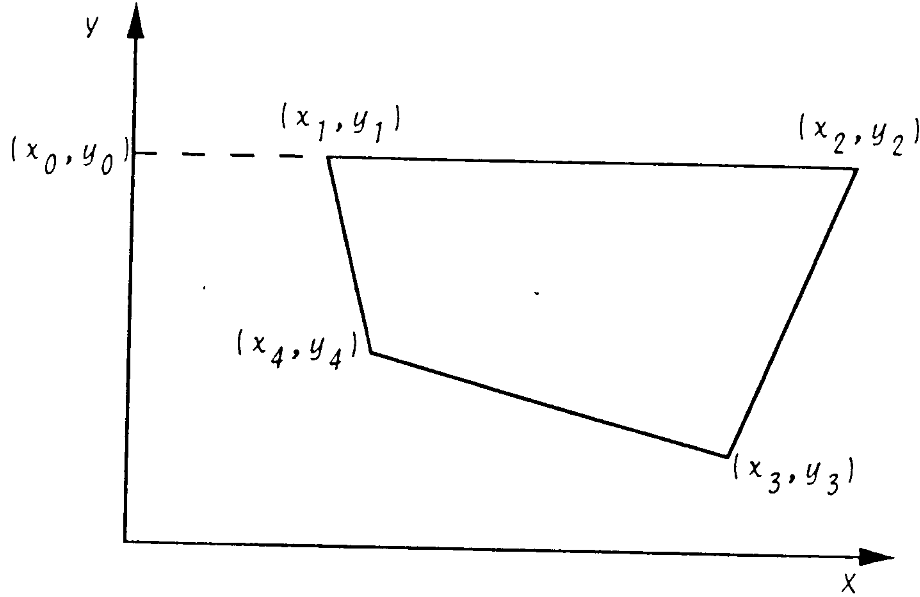


Figure 2.4: Planar graph embedded in the plane with $b = 0$.

here under line segment $(x_0, y_0) - (x_1, y_1)$ gives

$$m_{02} = \int_0^{x_1} \int_0^{y_1} y^2 \, dy \, dx = \int_0^{x_1} \frac{1}{3} y^3 \, dx \tag{2.22}$$

Using the line segment equation $y = y_0 + bx$ with $b = 0$ results in

$$m_{02}^{1h} = \frac{1}{3} y_0^3 x_1 \tag{2.23}$$

A similar derivation for line segment $(x_0, y_0) - (x_2, y_2)$ leads to

$$m_{02}^{2h} = \frac{1}{3} y_0^3 x_2 \tag{2.24}$$

The difference $m_{02}^{2_h} - m_{02}^{1_h}$ gives the second order moment $m_{02}^{1,2_h}$ under the horizontal line segment $(x_1, y_1) - (x_2, y_2)$ as

$$m_{02}^{1,2_h} = \frac{1}{3}y_0^3(x_2 - x_1) = \frac{1}{3}y_1^3(x_2 - x_1) \tag{2.25}$$

since $y_0 = y_1 = y_2$. The general expression for horizontal line segments $(x_i, y_i) - (x_{i+1}, y_{i+1})$ can now be written as

$$m_{02}^{i,i+1_h} = \frac{1}{3}y_i^3(x_{i+1} - x_i) \tag{2.26}$$

The second order moments for a complete planar object represented by an embedded graph of even degree are computed by adding the contributions of all the edges, taking into account whether or not the edge is horizontal or vertical. After similar derivations to those shown above, the complete second and first order moments for a planar object are as follows:

$$m_{02} = \sum_{i=1}^{n} \begin{cases} \frac{1}{12b_i}(y_{i+1}^4 - y_i^4) & \text{for } b_i \neq 0 \\ \frac{1}{3}y_i^3(x_{i+1} - x_i) & \text{for } b_i = 0 \end{cases} \tag{2.27}$$

$$m_{20} = \sum_{i=1}^{n} \begin{cases} \frac{b_i}{12}(x_{i+1}^4 - x_i^4) & \text{for } b_i \neq \infty \\ \frac{1}{3}x_i^3(y_{i+1} - y_i) & \text{for } b_i = \infty \end{cases} \tag{2.28}$$

$$m_{01} = \sum_{i=1}^{n} \begin{cases} \frac{1}{6b_i}(y_{i+1}^3 - y_i^3) & \text{for } b_i \neq 0 \\ \frac{1}{2}y_i^2(x_{i+1} - x_i) & \text{for } b_i = 0 \end{cases} \tag{2.29}$$

$$m_{10} = \sum_{i=1}^{n} \begin{cases} \frac{b_i}{6}(x_{i+1}^3 - x_i^3) & \text{for } b_i \neq \infty \\ \frac{1}{2}x_i^2(y_{i+1} - y_i) & \text{for } b_i = \infty \end{cases} \tag{2.30}$$

$$m_{11} = \sum_{i=1}^{n} \begin{cases} \frac{(y_{i+1} - y_{0_i})^2}{24b_i^2}a_{i+1} - \frac{(y_i - y_{0_i})^2}{24b_i^2}a_i & \text{for } b_i \neq 0 \\ \frac{y_i^2}{4}(x_{i+1}^2 - x_i^2) & \text{for } b_i = 0 \end{cases} \tag{2.31}$$

where b_i is the slope of line segment $(x_i, y_i) - (x_{i+1}, y_{i+1})$, b_1 = the slope of line segment $(x_1, y_1) - (x_2, y_2)$, n = the number of vertices, vertex $n + 1 \equiv$ vertex 1, $a_{i+1} = (y_{0_i}^2 + 2y_{0_i}y_{i+1} + 3y_{i+1}^2)$ and $a_i = (y_{0_i}^2 + 2y_{0_i}y_i + 3y_i^2)$. Notice that the expression

for m_{11} necessarily involves the intersection y_0 of the ordinate axis with the line segment $(x_i, y_i) - (x_{i+1}, y_{i+1})$. For $b_i = \infty$, there is no intersection y_0, thus equation equation (2.31) becomes zero. The orientation of any planar object represented by the coordinates of edge vertices can be computed using these equations in conjuncti on with equations (2.4) and (2.5). The result of computing the orientation of an object using the three methods are tabularized in Table 2.1. Method I is a calculation of

Rotation	Computed Orientation				
	Method I		*Method II*		*Method III*
	Size image		*j*		
	256	512	256	512	
5	5.24	5.037	4.80	4.99	5.001
10	9.77	10.02	9.70	9.97	10.001
15	15.04	15.12	14.69	14.97	15.002
20	19.83	20.06	19.70	20.03	20.001

Table 2.1: The results of computing object's orientation.

the object's orientation using equation (2.6), method II using Simpson's method of equation (2.12) and method III using equations (2.28 - 2.31) in conjunction with equations (2.4) and (2.5).

2.2 Coordinate transformation

In general, a transformation for two-dimensional homogeneous coordinates is represented as [20]

$$[T] = \begin{bmatrix} a & b & p \\ c & d & q \\ m & n & s \end{bmatrix} \tag{2.32}$$

where a, b, c, and d produce scaling and rotation; p and q produce reflection and shearing; and m and n produce translation. The homogeneous coordinates of the nonhomogeneous position vector $\begin{bmatrix} x & y \end{bmatrix}$ are $\begin{bmatrix} x' & y' & h \end{bmatrix}$ where $x = \frac{x'}{h}$ and $y = \frac{y'}{h}$.

In the Cartesian plane, homogeneous coordinates of the form $\begin{bmatrix} x & y & 1 \end{bmatrix}$ are used to represent the vector position $\begin{bmatrix} x & y \end{bmatrix}$ with the homogeneous coordinate scale factor $h = 1$. For all affine transformations in the Cartesian plane with $h = 1$, the values of p, q in the matrix transformations are set to be 0 and the value of s is set to be 1. Equation (2.32) becomes

$$[T] = \begin{bmatrix} a & b & 0 \\ c & d & 0 \\ m & n & 1 \end{bmatrix} \tag{2.33}$$

Translating objects in the Cartesian plane from one position to another position can be accomplished by properly assigning the values of $a = d = 1$, and $b = c = 0$ in equation (2.33).

Scaling of an object can be done by properly assigning the scaling factors into a and d. The scaling factors greater than 1 causes the object to be enlarged; reduction of an object is achieved by assigning the scaling factors less than 1.

By substituting $a = \cos\theta$, $b = \sin\theta$ $c = -\sin\theta$, $d = \cos\theta$ and $m = n = 0$ into equation (2.33), the result is a transformation matrix $R^{obj}_{(0,0)}$ rotating the objects about the origin through an arbitrary angle θ in a fixed coordinate system. This rotation matrix is represented as [20]

$$\left[R^{obj}_{(0,0)}\right] = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.34}$$

To rotate the object about an arbitrary point (m, n) these following transformations are performed :

- Translation of the object to the origin.
- Rotation of the object about the origin.
- Translation of the object back to the original center of rotation.

Concatenating these three transformation matrices, the rotation matrix $R^{obj}_{(m,n)}$ for rotating the object about an arbitrary point (m, n) is represented as [20]

$$\left[R^{obj}_{(m,n)}\right] = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ \{-m\ (\cos\theta - 1) + n\ \sin\theta\} & \{-n\ (\cos\theta - 1) - m\ \sin\theta\} & 1 \end{bmatrix} \tag{2.35}$$

This transformation matrix performs transformations of all vertices belonging to the object into another set of vertices in a fixed coordinate system. It may be desirable to consider transformations as a change of coordinate system. The rotation matrix performing a rotation of coordinate systems is represented as [20]

$$[R^{cs}(\theta)] = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.36}$$

The translation and rotation of coordinate systems is necessarily involved in the process of converting planar graphs of even degrees into OAQ's. In this case, a right hand coordinate system is used, θ indicates the counter-clockwise rotation, and the negative value of θ rotates the object in a clockwise direction. Consider the object depicted in Figure 2.5. This object has orientation θ with respect to the $x_1 y_1$ coordinate system (system 1). Once the orientation, θ, and the centroid coordinates (x_{cd}, y_{cd}), of the object have been determined, we wish to express the new coordinate system aligned with the object's principle axis of inertia. A transformation of coordinate system is necessary. By defining T_{12} as a transformation from coordinate system 1 to coordinate system 2, the transformation matrix can be represented as

$$T_{12} = T(-x_{cd}, -y_{cd})\ R(\theta) \tag{2.37}$$

where $T(-x_{cd}, -y_{cd})$ is a transformation matrix performing a translation of coordinate system 1 to a intermediate coordinate system (shown as dashed lines in Figure 2.5)

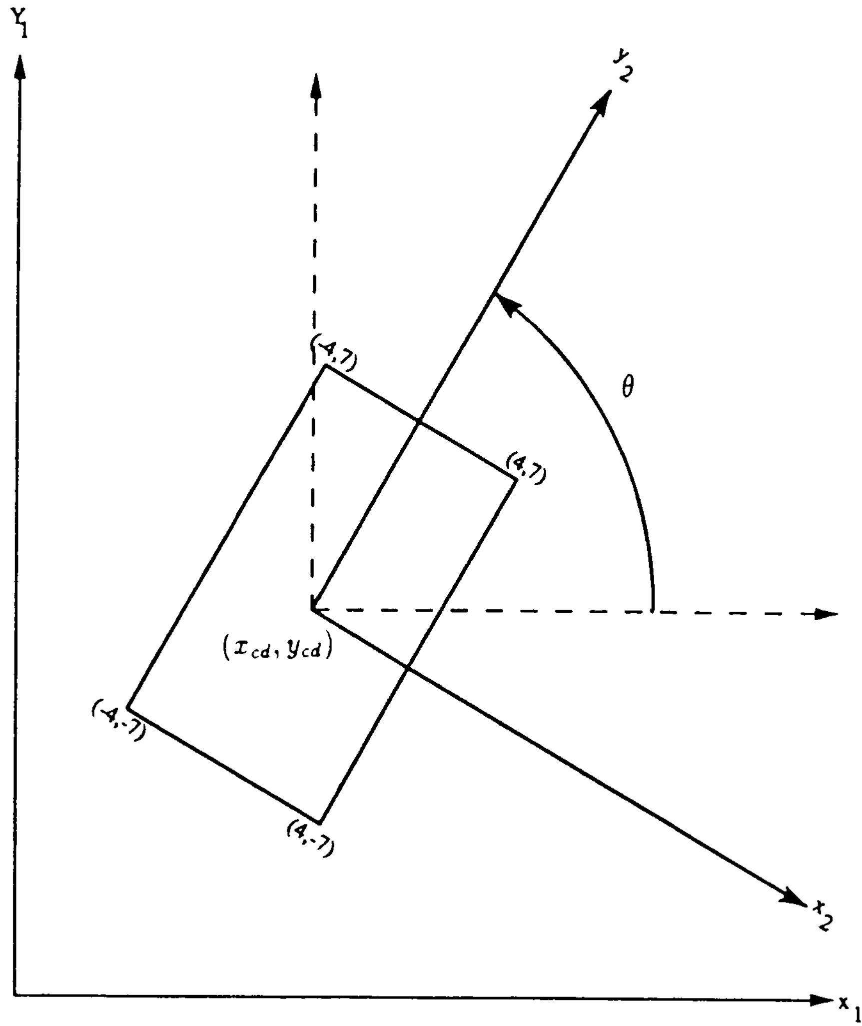


Figure 2.5: Coordinate systems 1 (original) and 2 (object space).

with the origin centered at the centroid of the object. This translation matrix is defined by substituting $m = -x_{cd}$ and $n = -y_{cd}$ of equation (2.33), and the rotation matrix $R(\theta)$ is represented as in equation (2.36). The rotation matrix performs a rotation of the intermediate coordinate system about the centroid of the object by an arbitrary angle θ. The result of these transformation is a coordinate system 2 shown in Figure 2.5. To represent all edge vertices in coordinate system 2, the transformation is performed using T_{12}, as follows :

$$[T_{12}] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -x_{cd} & -y_{cd} & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.38}$$

By carrying out the matrix product, we can rewrite equation (2.38) in the form

$$[T_{12}] = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ (-x_{cd}\ \cos\theta - y_{cd}\ \sin\theta) & (x_{cd}\ \sin\theta - y_{cd}\ \cos\theta) & 1 \end{bmatrix} \tag{2.39}$$

In our discussion, we consider each object as a planar graph represented by a set of points, defined by $P_i(i = 1, 2, 3, \cdots)$ whose coordinates are given in coordinate system 1, then P_i' can be rewritten as

$$P_i' = P_i \quad T_{12} \tag{2.40}$$

Every point belonging to an object now has its position in the new coordinate system (coordinate system 2).

For further discussion, coordinate system 2 in Figure 2.5 is called the *coordinate system in object space*. Conversion of planar graphs (objects) to OAQs needs to transform all edge vertices from *object coordinates* to *image coordinates*. Image coordinates are used to represent quadtree block corners. Before converting the vertices from object coordinates into image coordinates, we need to specify the size of the object space and the size of the image space.

The size of the object space $sizo$ is chosen to be the smallest power of two larger than the coordinate range belonging to the object. The size of image space $sizi$ is assigned depending on the required resolution. The situation shown in Figure 2.5 requires $sizo = 16$.

Once the size of the object space and the size of the image space has been given, all vertices belonging to the object can be converted from object coordinates to image coordinates using the transformation equation

$$\left[P_i''\right] = \begin{bmatrix} x_i'' \\ y_i'' \end{bmatrix} = ratio \begin{bmatrix} x_i' + \frac{sizo}{2} \\ y_i' + \frac{sizo}{2} \end{bmatrix} \tag{2.41}$$

where (x_i'', y_i'') are coordinates in coordinate system 3, also known as image space, and (x_i', y_i') are coordinates of point i in coordinate system 2. The scale factor is defined as $ratio = \frac{sizi}{sizo}$. Figure 2.6 shows the effect of the coordinate transformation, with two spaces involved; i.e. the original object space and the differently oriented image space. One edge of the original object is also shown. The dashed line represents unit increments in the image space coordinate frame.

Once the planar graph has been converted from object coordinates to image coordinates using orthogonal transformations (equations (2.40) and (2.41)), the raster representation of the graph in the image space can be generated using the Bresenham line drawing algorithm [6,19].

2.3 Representing a tightly closed boundary

The technique for representing unambiguous region boundaries and contour maps in raster format was developed by Merrill[15]. This method is based on a well-known topological property of a closed boundary; that is, in the case of discrete data, a point (x, y) is inside the region if and only if there exists a test line drawn from a point outside the region boundary to point (x, y) that crosses the boundary an odd number of times. This property has three restrictions [15,17] as follows :

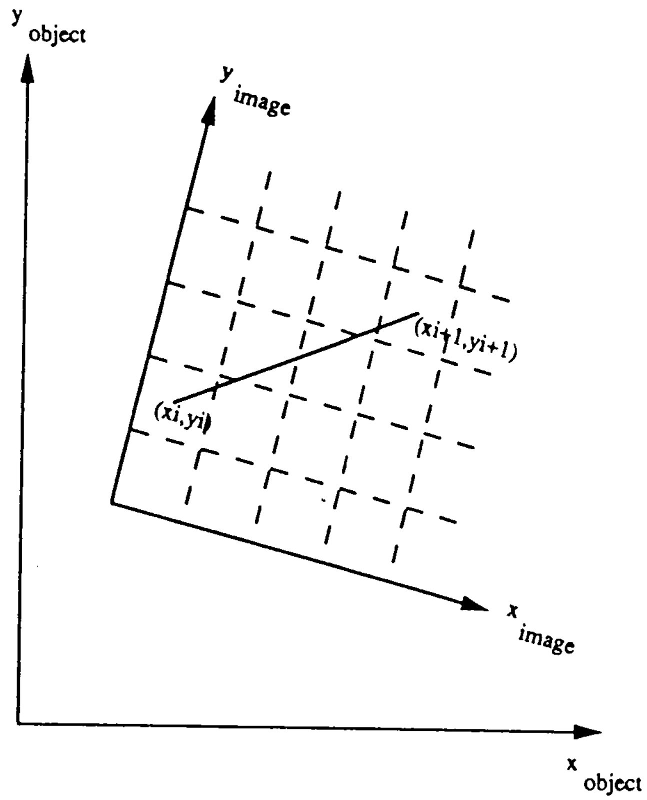


Figure 2.6: Object space and image space.

1. The boundary must be continuous and closed, in the sense that the maximum absolute distance between successive points in the locus must be equal to the grid element diagonal.

2. Provisions are necessary where the test line intersects the boundary tangentially.

   - Constrain the test line to be horizontal.

   - Augment the boundary at any local maximum and minimum, so that a tangential horizontal line passes through an even number of boundary points at each extremum.

   - At inflection points, augment the boundary to ensure that the test line passes through an odd number of points.

3. The closed boundary cannot intersect itself except that it must end at its starting point.

A tightly closed boundary (TCB) is a boundary meeting the above restrictions, so that continuous points in the locus are connected in the grid and it has been augmented at extrema and inflections. The TCB, in the manner described above, is necessary for converting a planar graph into OAQs. Once an initial boundary is computed using Bresenham's algorithm, the TCB is computed, which allows the formation of a run-length coded (RLC) version of the closed boundary and its interior.

Figure 2.7 illustrates finite point loci representing the closed boundary of an object. This representation is produced using Bresenham's algorithm. This figure shows the points between A and B do not meet restriction 1, thus the "missing" coordinates must be inserted in the locus. Points 1, 2 and 3 in the figure are examples where points are repeated in order to meet restriction 2.

Using the three restrictions, a set of points generated by Bresenham's algorithm may be augmented to ensure that the topological property of a closed boundary is fulfilled. The state diagram of an algorithm to generate the TCB is depicted in

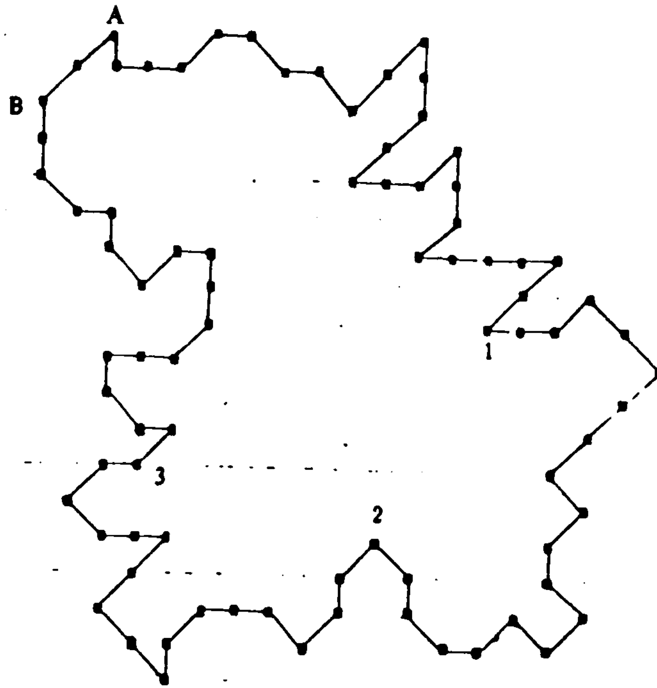


Figure 2.7: A closed boundary example.

Figure 2.8, along with the algorithm shown in Figure 2.9. The algorithm uses the

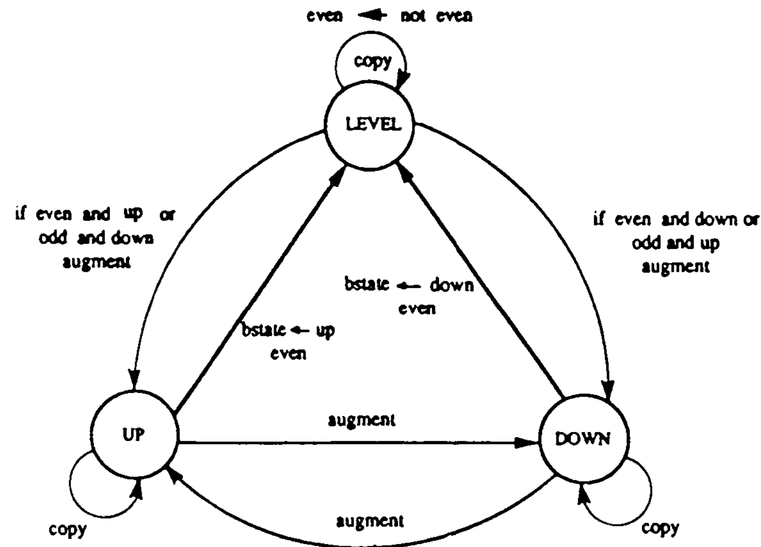


Figure 2.8: State diagram of an algorithm generating TCB.

unaugmented, closed boundary as input. To generate a run-length coded version, which will be described in the next section, the TCB is sorted using the y coordinate as the principal sort field.

2.4 Run Length Code (RLC) representation

One of many techniques which can compress the data for binary image representation is run-length coding. This method makes use of the fact that along any particular scan line there will be long runs of zeroes or ones. The length of such runs can be transmitted as numbers instead of transmitting the individual bits. There are several techniques for representing run-length code. For the image line

0111110011000

the run-length code can be [1,5,2,2,3]. The odd positions represent the number of zeroes in the image line, and the even positions represent the number of ones in the

```
dots = record
    y,x : integer; end

procedure Augment(ob : dots; nob : integer; var ab : dots;   var nab : integer);
{ Augment the grid cell boundary of a closed polygon }
var even,up : boolean; { Flags for count and direction }
    i,j,k,x1,y1,x2,y2 : integer
begin
    x1 := ob[1].x; y1 := ob[1].y; y2 := y1; i := nob + 1;
    {Trace backwards until a change in ordinate is found}
        while (y2 = y1) and (i > 0) do  begin
        i := i - 1; y2 := ob[i].y; end;
    k := i; { Marker or point of ordinate is found }
    if y2 < y1 then up := true else up := false;
    if (( nob - k) mod 2) = 0 then even := true;  else even := false;
    { Copy these initial cells to the augmented list }
    j := 0;
    for i := k + 1 to nob do begin j := j + 1;   ab[j] := ob[i]; end;
    j := j + 1; ab[j] := ob[1];
    { Check the rest of the boundary }
    for i := 2 to k do begin
        x2 := ob[i].x;   y2 := ob[i].y;
        if y2 > y1 then begin
            if (not even and not up)  or (even and up) then begin
                j := j + 1; ab[j].x := x1; ab[j].y = y1;  end;
            even := true; up := false; end
        else if y2 < y1 then begin
            if (not even and up) or (even and not up) then begin
                j := j + 1;  ab[j].x := x1; ab[j].y := y1; end
            even := true; up := false; end
        j := j + 1; ab[j].x := x2; ab[j].y := y2;
        even := not even; x1 := x2; y1 := y1;
    end
    { Check the last (k'th) position for augmentation }
    if k = nob then y2 := ob[1].y;  else y2 := ob[k+1].y;
    if (y2 > y1 and not up and not even)  or
        (y2 > y1 and up and even) or
        (y2 = y1 and not even) then begin
        j := j + 1; ab[j].x := x1; ab[j].y := y1; end
    nab := j;
end; { of procedure AUGMENT }
```

Figure 2.9: The AUGMENT algorithm for augmenting the raster boundary of a closed polygon.

image line. The beginning of each line may be indicated by some special code, or a convention that the line starts always with a zero or a one.

Run-length codes can be used to compute the area of the image. We use the convention that r_{ij} is the jth run of the ith line, and the first run of each row is zeroes. Therefore all the even runs correspond to ones in the image. Supposing that m_i is the number of runs on the i_{th} line, then this number is odd if the last runs contains zeroes.

The area of the image is represented by the sum of the run lengths corresponding to ones in the image.

$$A = \sum_{i=1}^{n} \sum_{j=1}^{m_i/2} r_{i,2j} \tag{2.42}$$

This sum is over the even runs only.

Run-length encoding can be specified in the simple form by encoding data into two groups. The first is the intensity and the second is the number of successive pixels on the scan line with that intensity [19].

Intensity	Run Length

Figure 2.10 illustrates a discrete binary image, along with the encoding for scan line 1,5 and 8.

A more compact run-length code scheme can be formed by including the sequential number of a scan line, with the encoding consisting of two groups. The first group is the scan line number, and the second group contains the starting position of successive pixels with intensity one, followed by the end position of those successive pixels. The compact representation is shown in Figure 2.11. For the scan line 8 of the binary image shown in Figure 2.10, the RLC code is shown in Figure 2.12. The run-length encoding of binary images provides a data compression that is significant not only for saving storage space. It also saves transmission time for facsimile transmission where run-length encoding is used extensively.

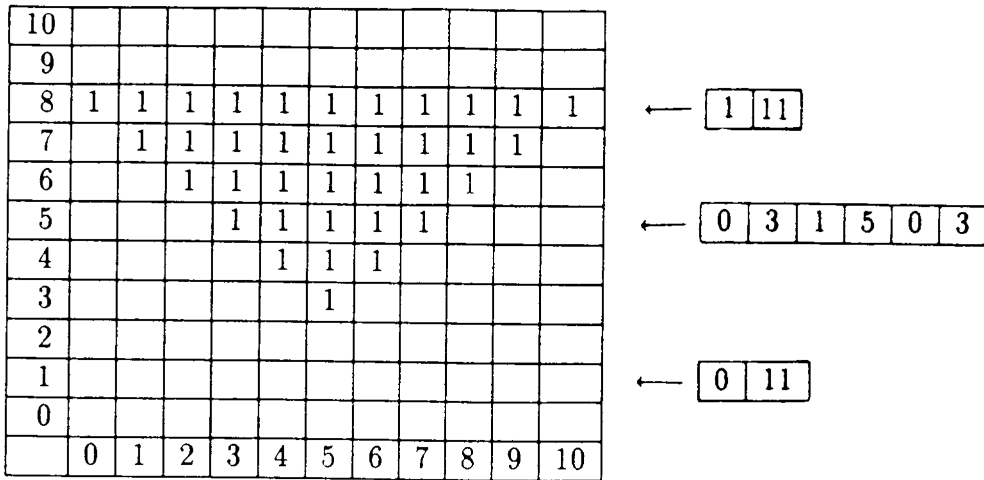


Figure 2.10: Example of RLC.

y_i	x_{begin}	x_{end}

Figure 2.11: The compact representation of RLC.

8	0	10

Figure 2.12: An example of RLC compact representation.

The previous section has presented all the details of conversion of the planar graph into a tightly closed boundary (TCB) representation. Once the TCB representation has been determined, a lexicographical sort using the y coordinate as the principal sort field is carried out to produce a compact run-length coded description of the graph. The run-length coded version is similar to the one shown in Figure 2.12. The algorithm for generating a run-length coded description of the graph is shown in Figure 2.13.

```
dots : record
    x, y : integer; end
rlength :record
    r, xb, xe : integer; end
rlc : array [1..max_runlength] of rlength;
nrlc : integer;

procedure TO_RLC (ab : dots; nab : integer);
var row,col1,col2,x,y,i : integer; odd : boolean;
begin
    i := 1; nrlc := 0; x := ab[1].x; y := ab[1].y;
    while i ≤ nab do begin
        row := y; col1 := x; odd := true;
        repeat
            col2 := x; i := i + 1; odd := not odd;
            x := ab[1].x; y := ab[i].y;
        until (y ≠ row) or ((x - col2) > 1) or (i > nab)
        nrlc := nrlc + 1; rlc[nrlc].r := row;
        rlc[nrlc].xb := col1; rlc[nrlc].xe := col2;
    end; {while}
end
```

Figure 2.13: The TO_RLC algorithm for generating a run-length coded (RLC) description of the graph.

2.5 Quadtree Encoding

Once a run-length coded description of the graph has been generated, the quadtree corresponding to it is generated using a modified Morton matrix approach [21]. To represent an object as an OAQ, a record structure termed **OAQ** is necessary. This is

a variant record with a fixed part and a variant part. The fixed part contains those fields that are always part of the record. The fixed part contains fields **QT**, **ORIENTATION** ,**CENTROID**,**SIZI**, and **SIZO** which are used to store a pointer to the root of the quadtree representing the object, the orientation of the OAQ, the object's centroid, the image space size and the object space size respectively. The variant part contains two fields, **POLY_COUNT** and **POLY_LIST**, which are used to store information belonging to polygons computed from boolean OAQ operations (see Chapter 3). These fields are required when boolean OAQ operations between two images are carried out. To determine whether these fields are required or not, the tag field is necessary. This tag field is declared as OP, and is set to be **true** to complete the record when boolean operations between OAQs are required. The Pascal record structure for an OAQ is given in Figure 2.14.

```
vertex = record
    x,y : real end;
pqtree = ^qtree; { pointer to a quadtree }
qtree = record { node of a quadtree }
    COLOUR : (B,W,G);
    NW,NE,SW,SE : pqtree;
    PARENT : pqtree end;
OAQ = record { Orientation Adaptive }
    QT : pqtree; { Quadtree }
    ORIENTATION : real;
    CENTROID : vertex;
    SIZI  : integer; {image size}
    SIZO  : integer; {object size}
    case OP : boolean of
        True :
            POLY_COUNT : integer;
            POLY_LIST : polygon_array
end;
```

Figure 2.14: Pascal record structure for an orientation adaptive quadtree.

To generate the quadtree, a modification of the Morton matrix was made. The decision about whether an image space point is interior to the object is inferred

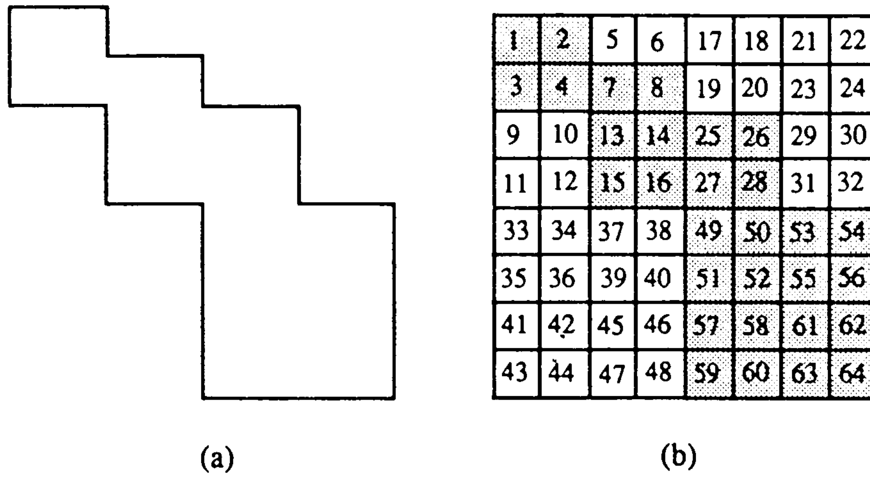
from the RLC. In Morton matrix fashion, the pixels are visited in the order shown in Figure 2.15. The leaf node is only created if it is maximal, and a merge occurs

1	2	5	6	17	18	21	22
3	4	7	8	19	20	23	24
9	10	13	14	25	26	29	30
11	12	15	16	27	28	31	32
33	34	37	38	49	50	53	54
35	36	39	40	51	52	55	56
41	42	45	46	57	58	61	62
43	44	47	48	59	60	63	64

Figure 2.15: Morton matrix.

if four sons of a node are either BLACK or WHITE. If all the pixels are not of the same type no further merging can occur, and the segment of the final quadtree corresponding to their contribution can be constructed. For example, the object shown in Figure 2.16 is a 2^3 by 2^3 array unit square of "pixels", each of which has value 0 or 1. Denoting this array by A, the element array A[1,1] is inspected first, followed by A[1,2], A[2,1],A[2,2], A[1,3] and so on. Since a leaf node is only created if it is maximal, pixels 1,2,3 and 4 are of the same type BLACK, and they will be represented as a node A in the final quadtree. As pixels 5,6,7 and 8 are not of the same type (pixels 5,6 are WHITE and 7,8 are BLACK), their nodes cannot participate in any further merge. The corresponding subtree that contributes to the final quadtree is shown in Figure 2.17. As an example for further merges, pixels 33-48, and 49-64 are ultimately represented as nodes H and I in the final quadtree. These nodes are created once their remaining siblings have been completely processed.

The same concept is adopted when converting objects into OAQs. Once the orthogonal transformations have been carried out, the unit increments associated with the oriented image space are similar to the unit squares shown in Figure 2.16. The object depicted in Figure 2.5 can be represented as the OAQ shown in Figure 2.18.



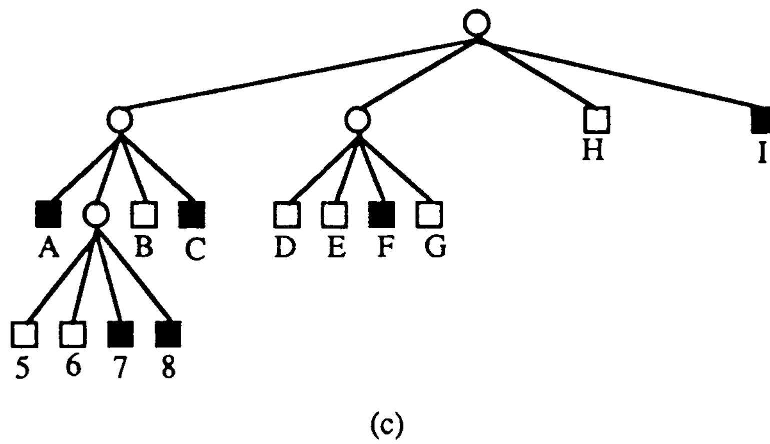


Figure 2.16: (a) Sample image. (b) Its block decomposition. (c) Quadtree representation of the blocks.

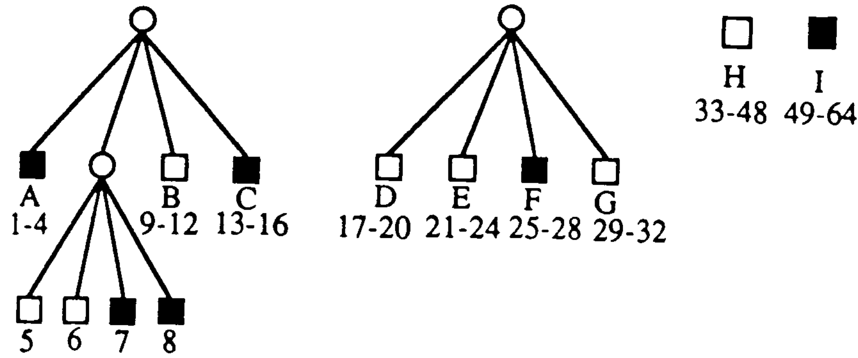
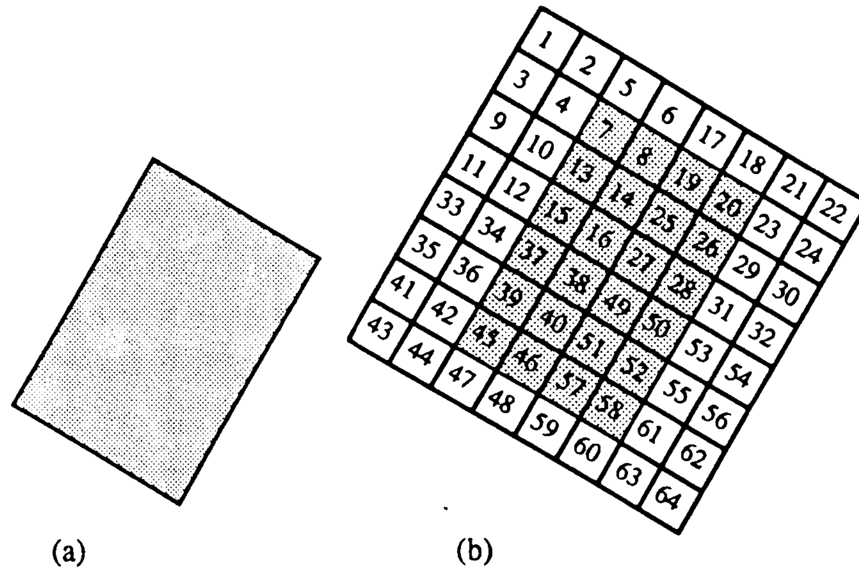


Figure 2.17: Subtree in the process of obtaining a quadtree corresponding to Figure 2.16(b).

The construction of a quadtree corresponding to the given object is performed by function QT_MAKE and procedure CONSTRUCT. Function QT_MAKE controls the construction of the final quadtree. It returns a pointer which points to the quadtree's root. If the image is all BLACK or WHITE, this function returns a one-node tree. The construction of the quadtree is actually accomplished by procedure CONSTRUCT. This procedure examines all the pixels in Morton matrix fashion. Each pixel is recursively inspected by inferring from the rlc description. If the image space point is interior to the object, this point is said to be BLACK; otherwise it is said to be WHITE. To examine a pixel, a function IMAGE, which indicates whether a pixel is BLACK or WHITE, is recursively invoked by using the values of northeasternmost pixel coordinates. If these coordinates are in the run-length coded description, a function IMAGE returns BLACK, otherwise it returns WHITE. The procedure CONSTRUCT recursively examines all pixels and creates nodes whenever all four sons are not of the same type. Procedure CONSTRUCT returns the result of inspecting sons, which is stored in a record called **PAIR** containing the two fields termed **COLOUR** and pointer **POINT**.

Once the process of constructing the OAQ has been completed, it is identical to an ordinary quadtree, except for the storage of the orientation , the centroid, the image



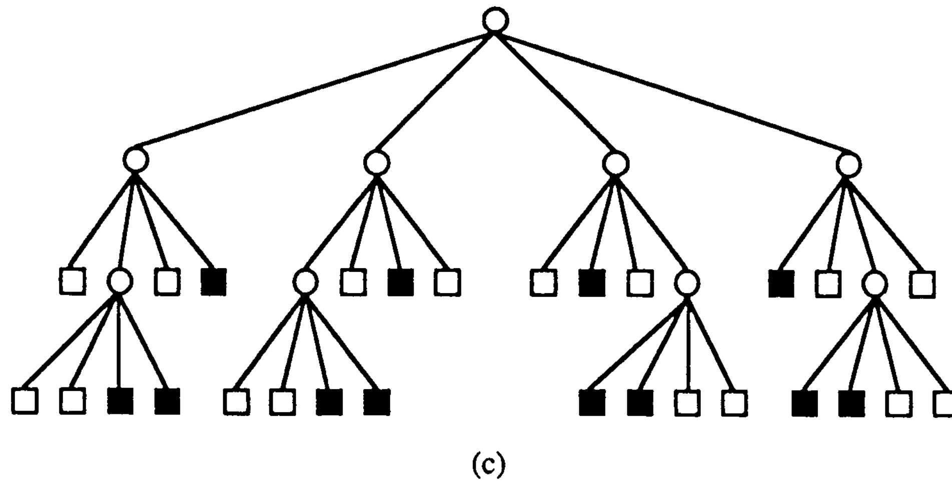


Figure 2.18: Construction of an OAQ (a)Sample oriented image. (b)Its block decompositions. (c)OAQ representation of the blocks.

size space, and the object size space at the root, and the variant storage required to carry out boolean OAQ operations. This quadtree is stored in the field of a record **OAQ** which is labelled **QT**, and can be traversed in preorder traversal to produce a set of codes which will be stored on disk along with the orientation of the object, the object's centroid, the image size space, and the object size space. 'B' is assigned to be the code for a BLACK node, 'W' is assigned to be the code for a WHITE node, and '(' represents a GRAY (i.e neither black nor white) node. This set of codes is called a linear code and it is helpful for display purposes. The linearly coded OAQ for representing the object in Figure 2.18 is shown in Figure 2.19. The functions and

((W(WWBBWB((WWBBWBW(WBW(BBWW(BW(BBWWW

Figure 2.19: Linear coding for the OAQ of Fig.2.19.

procedures, are shown in Figures 2.20, 2.21 and 2.22.

```
pair = record
    colour = char; point = pqtree; end;
function QT_MAKE(n:integer) : pqtree;
q : pqtree;
p : pair;
begin
    CONSTRUCT(p,n,2^n,2^n);
    if p.colour = 'G' then begin
        p.point^.parent := nil;
        QT_MAKE := p.point; end
    else begin
        new(q); q^.colour := p.colour;
        q^.NW := q^.NE := q^.SW := q^.SE := nil;
        q^.parent := nil; QT_MAKE := q;
    end;
end; { of procedure QT_MAKE }
```

Figure 2.20: The QT_MAKE algorithm for controlling the final quadtree.

Converting a planar graph of even degree to OAQs can be summarized as follows:

1. Calculate the orientation of the object using the method described in section 2.1.

```
rlength = record
    r, xb, xe : integer; end
rlc : array [1..max_runlength] of rlength;

function IMAGE (x,y : integer) : integer;
i : integer;
begin
    i := 1;
    while ((rlc[i].r < y) and (i ≤ nrlc)) do i := i + 1;
    if ((rlc[i].r ≠ y) or (i > nrlc)) then IMAGE := 0
    else begin
        while (rlc[i].r = y) and (rlc[i].xe < x) do i := i + 1; end
    if ((rlc[i].r = y) and (rlc[i].xb ≤ x) and and (rlc[i].xe ≥ x)) then
        IMAGE := 1 else IMAGE := 0;
end {of function IMAGE}
```

Figure 2.21: The IMAGE algorithm for determining if image pixel (x,y) is BLACK or WHITE.

2. Convert all edge vertices from object coordinates to image coordinates using orthogonal transformations; equations (2.40) and (2.41).

3. Use the Bresenham algorithm [19] to obtain a raster representation of the graph in image space.

4. Obtain the tightly closed boundary (TCB) representation of this raster representation using the augmentation algorithm of [15]. This ensures topological consistency in that one can always infer directly from the TCB whether a point is interior to the object represented by the graph or not.

5. Sort the TCB lexicographically using the y coordinate as the principal sort field. This produces a run-length coded (rlc) description of the graph.

6. Using this rlc description, generate the quadtree corresponding to it using a modified Morton matrix approach [21]. The modification made here is that the decision about whether an image space point is interior to the object is inferred

```
procedure CONSTRUCT(var par : pair; n,x,y : integer)
var ; p: array[1..4] of pair; r,q : pqtree; level,i,k : integer;

begin
    if (n = 0) then begin { if block is a single pixel }
        if IMAGE(x-1,y-1) = 0 then par.colour := 'W' else par.colour := 'B';
        par.point := nil; end
    else begin { if block contains more then 1 pixel }
        level := n-1;
        CONSTRUCT(p[1],level,x-2^level,x-2^level); CONSTRUCT(p[2],level,x,y-2^level);
        CONSTRUCT(p[3],level,x-2^level,y); CONSTRUCT(p[4],level,x,y);
        if (p[1].colour ≠ 'G') and (p[1].colour = p[2].colour) and (p[2].colour = p[3].colour) and
        (p[3].colour = p[4].colour) then par := p[1]; { creates a leaf node }
        else begin { creates subtree }
            new(q);
            for i := 1 to 4 do begin
                if p[i].colour = 'G' then begin
                    case i of
                        1 : q^.NW := p[1].point; 2 : q^.NE := p[2].point;
                        3 : q^.SW := p[3].point; 4 : q^.SE := p[4].point;
                    end; { case }
                    p[i].point^.parent := q;
                end { if }
                else begin  { p[i].colour ≠ 'G' }
                    new(r); r^.colour := p[i].colour;
                    r^.NW := r^.NE := r^.SW := r^.SE := nil;
                    { initialization of polygon list for Boolean OAQ operation }
                    if p[i].point^.OP = true then begin
                        r^.polygon_count := 0; { initialize a counter of the number of polygons }
                        for k := 1 to max_poly do begin
                            r^.polygon_list[k].history[1] := r^.polygon_list[k].num_links := 0;
                            r^.polygon_list[k].edge_list :=nil; end; { for k }
                    end {if OP is true} ;
                    case i of
                        1 : q^.NW := r; 2 : q^.NE := r; 3 : q^.SW := r; 4 : q^.SE := r;
                    end; { case }
                    r^.parent := q;
                end; { else }
            end; { for i}
            q^.colour := 'G';
            q^.polygon_count := 0;
            for k := 1 to max_poly do begin
                q^.polygon_list[k].history[1] := q^.polygon_list[k].num_links := 0;
                q^.polygon_list[k].edge_list := nil; end; { for k }
            par.point := q;
            par.colour := 'G';
        end; { else creates subtree }
    end; { else if block contains more than 1 pixel }
end; { of procedure CONSTRUCT }
```

Figure 2.22: The CONSTRUCT algorithm for constructing quadtree.

from the rlc description. This obviates the requirement to store a binary array of size (m, m), where m is the maximum image space coordinate.

The orientation adaptive quadtree takes advantage of large eccentricity in an object and accounts for any rotation of the principal axis of inertia. In this sense, it provides an optimal (in terms of space) quadtree encoding of connected planar objects. Computing the second order moments necessary to obtain the orientation requires $O(n)$ time, for n = number of edges in the original object description [18]. The extra time does not add significantly to the overall time of converting edge structures to quadtree structures which is $O(n^2)$, for n = image space coordinate range [18].

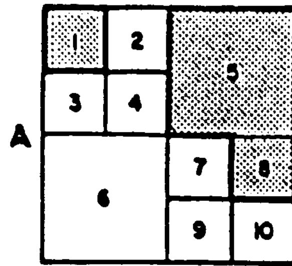
Chapter 3

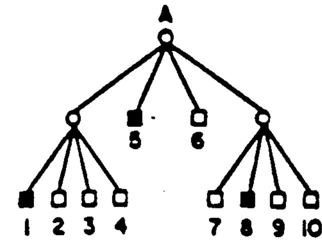
Boolean operations with OAQs

The quadtree structure is especially efficient when set operations such as intersection and union of two images are required. Determining the quadtree corresponding to the intersection of quadtrees A and B simply requires a traversal of two quadtrees in parallel [21]. Consider quadtrees A and B shown in Figure 3.1. For the intersection operation, construction of the resulting quadtree, called I, can be computed in such a way that if either of the two nodes is WHITE then the corresponding node in I is WHITE. If one node is BLACK, then the node in I is set to the corresponding node of the other quadtree. If both nodes are GRAY then the corresponding node in I is set to be GRAY, and the algorithm is applied to the children of A and B. Once all the children of the visited nodes have been processed (when both nodes are GRAY), a check must be made if a merger is to take place since all four children in I could be WHITE. For the union operation of two quadtrees, the algorithm is applied identically except that the role of BLACK and WHITE nodes are interchanged. The boolean operation of two standard quadtrees is depicted in Figure 3.1.

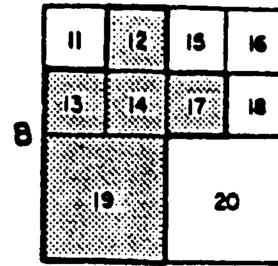
Intersecting two OAQs is not similar to the manner of intersecting two standard aligned quadtrees. The process of intersecting two OAQs can be divided into two steps as follows :

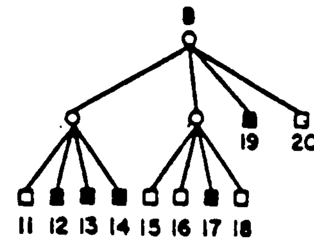
1. Extracting polygons by intersecting every two blocks representing two nodes



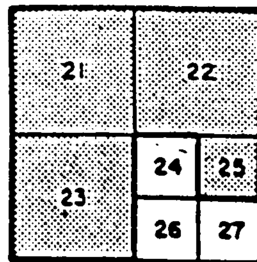


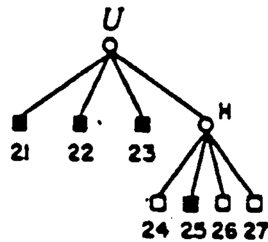
(a) Sample image and its quadtree.



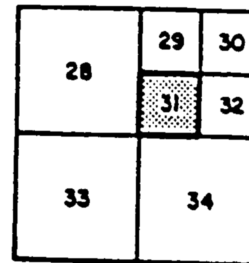


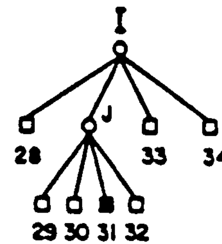
(b) Sample image and its quadtree.





(c) Union of the images A and B





(d) Intersection of the images A and B

Figure 3.1: Intersection of two standard quadtrees (from [21]).

from different OAQs. For the intersection operation, these two blocks represent two BLACK nodes, and for the union operation the two blocks represent a WHITE node of OAQ A and a BLACK node of OAQ B. The union operation also extracts the BLACK blocks of OAQ A.

2. Merging those polygons which result in new objects.

To clarify boolean operations of two OAQs, in the manner mentioned above, we shall now present the algorithm for performing boolean operations on two OAQs.

3.1 Definition and notation

Let the OAQ be stored as a record **QT** containing 6 fields. The last fields contain a pointer to an OAQ structure which is shown in Figure 2.14. Each node in the OAQ structure is stored as a record containing six fields. The last five fields contain pointers to the node's father and its four children labelled NW, NE, SW, SE, and PARENT. The QT record is stored as the first field in the record OAQ. For the reference OAQ A, each node has two additional fields which are labelled **POLY_COUNT** and **POLY_LIST**. As mentioned above, **POLY_LIST** is used to store polygons resulting from the intersection between two blocks representing black nodes belonging to OAQ A and B. The number of polygons in the list is stored in **POLY_LIST**. The Pascal record structure for the **POLY_LIST** and **POLY_COUNT** is shown in Figure 3.2. After constructing two OAQs, the orientation of each object is stored in a field labelled **ORIENTATION**, the centroid is stored in a field labelled **CENTROID**, the image space size is stored in a field labelled **SIZI**, and the object space size is stored in a field named **SIZO**.

To compute the intersection of two OAQs, we carry out the two steps mentioned above. These are discussed in more detail in the next section.

```
type
    abyte = array[1..max_level] of byte;
    edge_ptr = ^polygon_edge;
    polygon_edge = record { edge of polygon }
        vert1,vert2 : vertex; edge_ID : packed array[1..2] of char; level : integer; next : edge_ptr;
    end;
    polygon = record
        history : abyte; depth : integer; edge_list : edge_ptr ; end;
    polygon_array = array[1..max_poly] of polygon;
    poly_list : polygon;
```

Figure 3.2: The Pascal record structure for Boolean OAQ operations.

3.2 Process of extracting polygons

This process extracts polygons produced by the intersections between two blocks, each representing nodes of two different OAQs, A and B. For the intersection OAQ operation, the two blocks represent BLACK nodes. For the Boolean union of two OAQs, the polygons come from black blocks of OAQ A and the intersections between blocks representing WHITE nodes of OAQ A and BLACK nodes of OAQ B. This process requires a record for four corner points of every block of OAQ B in the OAQ A reference frame. The computation of corners points of the block involves a rotation and translation operation, which is explained in the following subsection. Extracting polygons requires the examination of nodes of OAQs A and B by traversing them in parallel starting from their roots. If both blocks representing the root nodes of A and B intersect, then an examination between each child of the root of node A and all children of the root node of B is carried out recursively. If the two blocks representing the root nodes of OAQs A and B do not intersect, then no operations are performed since the two OAQs are disjoint.

Boolean OAQ operations require a test to decide whether two blocks representing the specified nodes of OAQ A and B intersect or not. For the intersection operation the specified nodes are BLACK nodes of OAQ A and OAQ B. If the two blocks intersect, the resulting intersection is a polygon, which is stored in the node belonging to

the reference OAQ A. The way to extract the polygon will be explained in subsection 3.2.2. If the blocks being examined are coincident, then one of them is extracted as a result of the intersection. If the blocks are completely disjoint, then no resulting intersection is extracted. When the examined block of OAQ A lies inside the examined block of OAQ B, then the result is the complete node of OAQ A.

For the Boolean intersection, if both examined nodes are BLACK, then a test is carried out to see if the corresponding blocks intersect. If one of the nodes is GRAY and the other is BLACK, then the process described above is carried out recursively to the children of these nodes. If both nodes are GRAY, then the same process is carried out recursively to the children of both OAQs. If one of the nodes is WHITE, then no action is taken.

For the Boolean union, when the process of examining two nodes of OAQ A and OAQ B visits a BLACK node of OAQ A, then the block representing this node is directly stored in the node of OAQ A. The block of OAQ A is always extracted in the union operation of two blocks representing the BLACK nodes of OAQs A and B. When the process visits a WHITE node of OAQ A and a BLACK node of OAQ B, the intersection of the corresponding blocks is taken, since the intersection contributes to the result of the union. When a WHITE node of OAQ A and a BLACK node of OAQ B is found and their blocks intersect or coincident, then the resulting polygon is extracted and stored in the node of OAQ A.

When a polygon resulted from the operation of two blocks has been extracted, it is stored in a node of reference OAQ A, along with the path where the blocks representing a node of OAQ B came from. The process of storing the polygon will be discussed further in subsection 3.2.3.

After extracting all polygons of intersection between two blocks representing nodes of OAQs A and B, and having the results stored in reference OAQ A, OAQ B is no longer considered. Reference OAQ A is now traversed in post order, and a merge operation is carried out to merge all polygons stored in the leaf nodes of the reference

OAQ A. For the intersection operation, the polygons are stored only in the BLACK leaf nodes of the reference OAQ. For union operation, the polygons are stored not only in the BLACK leaf nodes, but also in the WHITE leaf nodes of OAQ A.

The result of this merge process is polygons which are stored in the node of the reference OAQ. When a process of traversing the OAQ visits a GRAY node, a merge process is carried out to merge all polygons stored in its children nodes. This process is done recursively up to the root. After finishing the traversal, the root node will have the resulting polygons of two intersecting OAQs. These objects can be represented into OAQs by transforming back to object coordinate space, calculating the orientation of the objects and forming the OAQs as described in chapter 2.

3.2.1 Recording a block of OAQ B in OAQ A

This process is intended to record corner points of a block represented by a black node of OAQ B in OAQ A's reference frame, using the information brought by OAQ B. Figure 3.3 shows two intersecting OAQs. The orientation of the object, the object's centroid, the image space size and the object space size are the information that are brought by OAQ.

Each block representing nodes of OAQ B can be known directly from its node position in the OAQ, but they are not known in OAQ A's reference frame. A calculation of the distance between centroid of A and centroid of B in A's reference frame is necessary to compute any block of OAQ B in the A reference frame. Figure 3.4 depicts blocks of OAQ A and OAQ B of Figure 3.3. Knowing A's and B's centroid in object common coordinate frame, the length of δ_x and δ_y shown in the figure can be calculated using the relation

$$\delta_x = r\cos(\alpha - \theta_A) \tag{3.1}$$

$$\delta_y = r\sin(\alpha - \theta_A) \tag{3.2}$$

where α is $\arctan(\frac{(y_{cd_B} - y_{cd_A})}{(x_{cd_B} - x_{cd_A})}$. The values of δx and δy are computed in the image

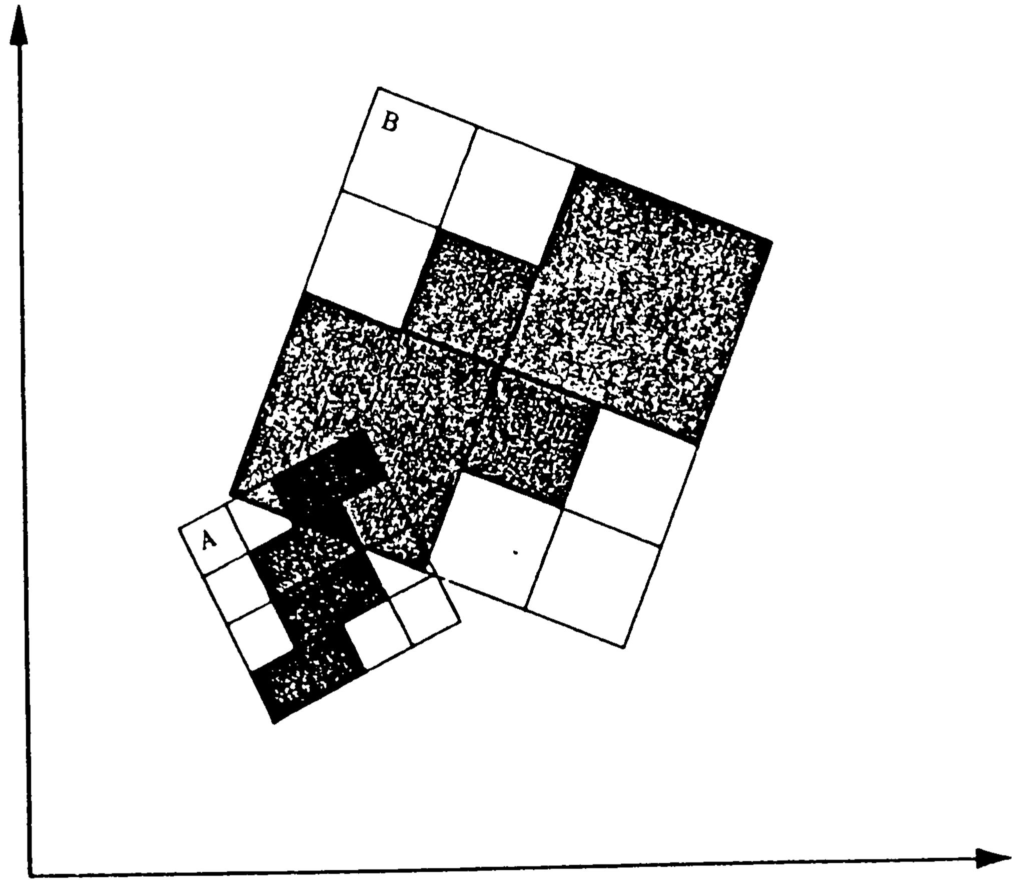


Figure 3.3: An example of the intersection of two OAQs.

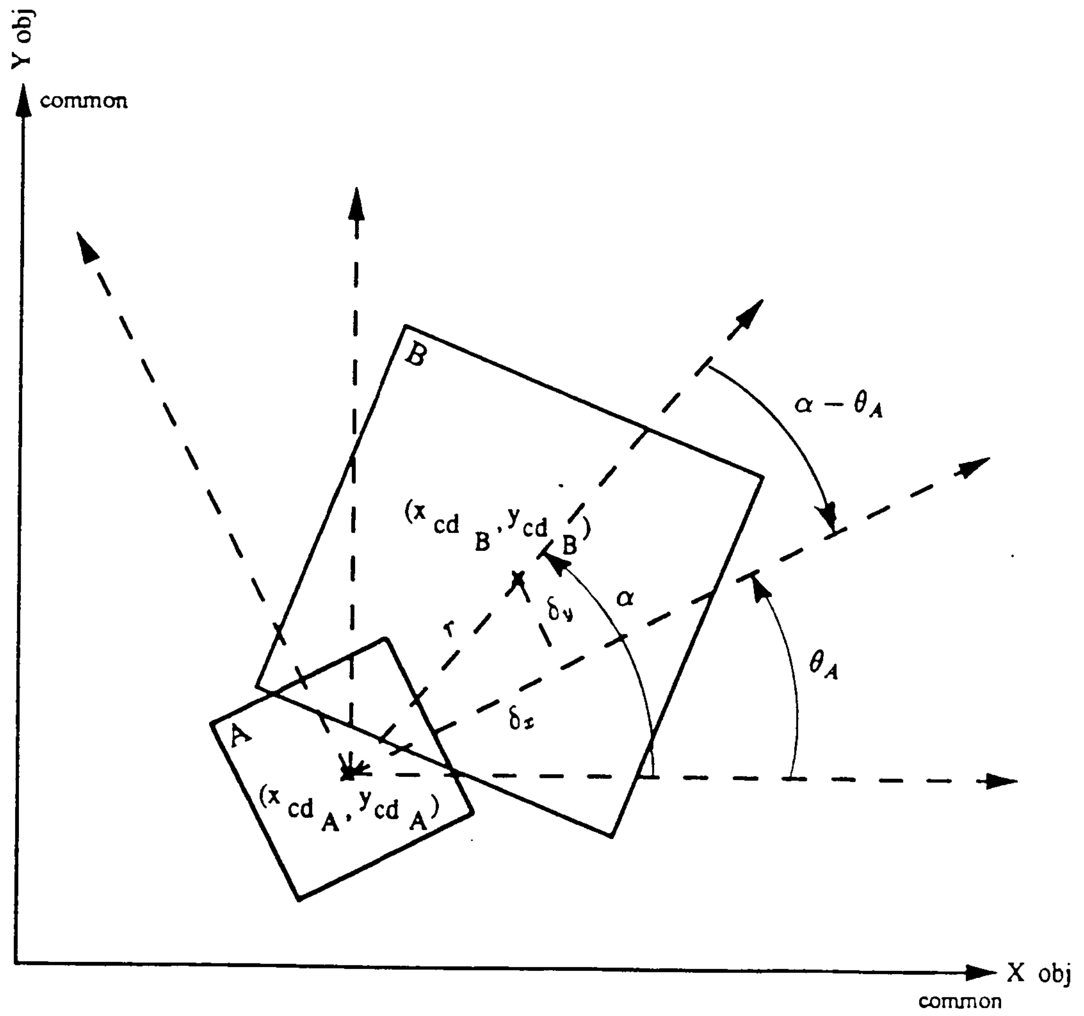


Figure 3.4: Outer blocks of OAQ A and B.

space coordinate of A by multiplying δx and δy with A's scale factor $ratio_A = \frac{sizi_A}{sizo_A}$. Knowing δx and δy, the computation of corner points of OAQ B in OAQ A requires a coordinate transformation from B's image space coordinate system to A's image space coordinate system. This is an orthogonal transformation accounting for the centroid difference $(\delta_x, \delta y)$, orientation difference $(\theta_{AB} = \theta_B - \theta_A)$ and scale difference $(ratio_{AB} = \frac{ratio_A}{ratio_B})$ for the two OAQs.

Two transformations are necessary to compute a corner point of a block of OAQ B in OAQ A's reference frame. The first is a scale transformation of the block of OAQ B by scale difference $ratio_{AB}$, and the second one is the rotation transformation of the block of OAQ B about its centroid, through θ_{AB}.

When the second transformation is carried out, the following transformations are involved [6].

Translation of the block to the origin.

Rotation of the block about the origin.

Translation of the block back to the original center of rotation.

In order to record a block of OAQ B in OAQ A's reference frame, the block is translated back to the center of OAQ B, which is $(C_x^B, C_y^B) = (0.5sizi_B, 0.5sizi_B)$. We take into account the fact that OAQ B's centroid is recorded in OAQ A's reference frame, which is $(C_x^{BA}, C_y^{BA}) = (0.5 \times sizi_A + \delta x, 0.5 \times sizi_A + \delta y)$.

By defining T_{BA} as a transformation from image space coordinate system B to A, the transformation matrix can be represented as

$$T_{BA} = S(ratio_{AB}) \;\; R(\theta_{AB}) \tag{3.3}$$

where $S(ratio_{AB})$ is a scaling matrix defined as in equation (2.33) with the values $a = d = ratio_{AB}$, and $R(\theta_{AB})$ is a rotation matrix defined as

$$[\mathrm{R}(\theta_{\mathrm{AB}})] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -C_x^B & -C_y^B & 1 \end{bmatrix} \begin{bmatrix} \cos\theta_{AB} & \sin\theta_{AB} & 0 \\ -\sin\theta_{AB} & \cos\theta_{AB} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ C_x^B & C_y^B & 1 \end{bmatrix} \tag{3.4}$$

By carrying out the matrix product in the homogeneous coordinates, equation (3.3) can be written as

$$[\mathrm{T_{AB}}] = \begin{bmatrix} ratio_{AB} & 0 & 0 \\ 0 & ratio_{AB} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta_{AB} & \sin\theta_{AB} & 0 \\ -\sin\theta_{AB} & \cos\theta_{AB} & 0 \\ -(C_x^B \cos\theta_{AB} + & -(C_x^B \sin\theta_{AB} - & 1 \\ C_y^B \sin\theta_{AB} + C_x^{BA}) & -C_y^B \cos\theta_{AB} + C_y^{BA}) & \end{bmatrix} \quad (3.5)$$

$$[\mathrm{T_{AB}}] = \begin{bmatrix} r_{AB}\cos\theta_{AB} & -\sin\theta_{AB} & 0 \\ \sin\theta_{AB} & r_{AB}\cos\theta_{AB} & 0 \\ -(C_x^B \cos\theta_{AB} + & -(C_x^B \sin\theta_{AB} - & 1 \\ C_y^B \sin\theta_{AB} + C_x^{BA}) & -C_y^B \cos\theta_{AB} + C_y^{BA}) & \end{bmatrix} \quad (3.6)$$

Let $P_i(i = 1 \cdots)$ be a set of corner points of any block of OAQ B. The point coordinates are known directly from the node position in the OAQ. Consider Figure 3.3. The two OAQs represent $sizi \times sizi$ binary images. The root of OAQ B represents a binary image of size 2^4x2^4. The coordinates of the upper right , upper left, lower left, and lower right corner points of the NE block of OAQ B are $(2^4, 2^4)$,$(2^3, 2^4)$, $(2^3, 2^3)$ and $(2^4, 2^3)$ respectively. These coordinates are in the OAQ B's image space. By applying equation (3.6) to P_i, then

$$P'_i = P_i \; T_{BA} \quad (3.7)$$

We now have a set of coordinates defining each block of OAQ B in OAQ A.

3.2.2 An arbitrarily oriented block intersection

The process of extracting polygons of two intersecting blocks is similar to a process of clipping one rectangular window against another rectangular window, except for defining the clipped polygon. Knowing the level of nodes from OAQ A and OAQ B, the position of blocks of A and of B can be properly calculated in A's reference frame. The process of intersecting two blocks representing a node of OAQ A and one of OAQ B is done by keeping track of the edges defining the polygon. Figure 3.5 depicts an intersection between two blocks that represent two black nodes. The position of four corner points of block A and block B have been recorded in A's

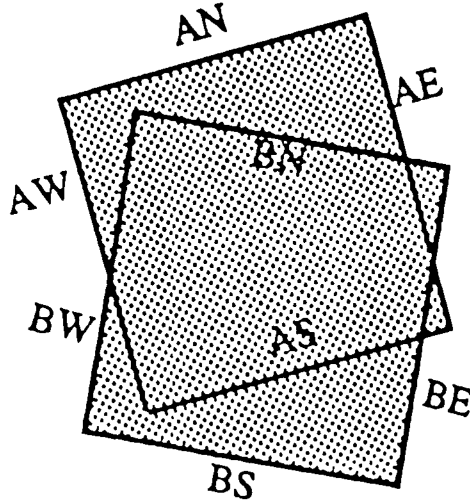


Figure 3.5: Intersection of two black blocks.

reference frame. The algorithm to extract the polygon is modified from Sutherland-Hodgman's algorithm [6] and Weiler's algorithm [34]. The output is designed to keep track of edges defining the polygon. Sutherland-Hodgman's algorithm accepts a series of polygon vertices and outputs another series of vertices defining the clipped polygon. The Weiler algorithm results in a set of vertices defining a polygon using a complicated Euler graph data structure, with each edge represented by a winged-edge data structure. The algorithm uses a set of successive vertices v_1, v_2, v_3, v_4 in order lower left, lower right, upper right and upper left of the corresponding nodes. The polygon edges are from v_1 to v_2, v_2 to v_3, v_3 to v_4 and v_4 to v_1. These edges are labelled respectively as AS, AE, AN, and AW for block A, and BS, BE, BN, and BW for block B. Each edge is taken associated with the position of the edge bounding the block; i.e, AE is an edge bounding block A on the East. To extract the intersection, a block of B is clipped against a single clip boundary. Successive clips against four boundaries are carried out to extract an edge list defining the result. The algorithm traces around block B from $v_1 \cdots v_4$, each step examining the relationship between two successive vertices and the clip boundary of block A, keeping track of the edges formed by those

two successive vertices. If one edge contributes to the clipped polygon, the endpoints and the edge names are added to the output list.

Figure 3.6 depicts the possible cases of two successive points clipped against a clip edge. In the case of an edge inside the clip boundary, (case 1), the two points form

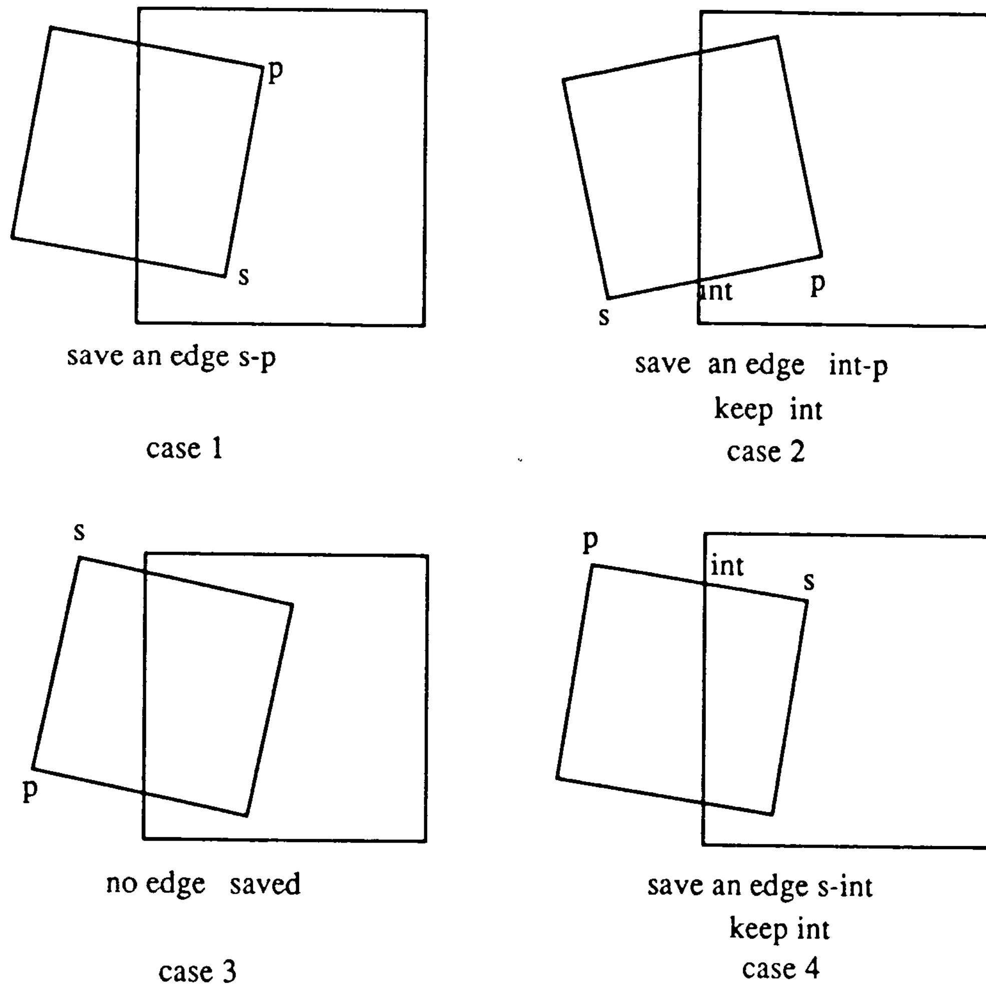


Figure 3.6: Four possible cases of clipping block.

an edge, and it is added to the output list along with the corresponding edge's name. In case 2, the polygon edges intersect the clip boundary, and the intersection point

and the endpoint which lies inside the clip boundary form an edge contributing to the clipped polygon. The two endpoints and the corresponding edge's name are added to the output list. It should be noted that if the intersection between the polygon edge and the clip boundary is found, then the intersection point is kept as a first point of one of the edges comprising the clipped polygon. A similar method is used for case 4; here the intersection point is necessarily kept as a second point of the edge. The appropriate name for this edge is easily identified by checking the relative position of its two vertices, and the clip edge. The complete block intersection algorithm is shown in Figure 3.7. Procedure ARBOR_BLOCK_INT depicted in the figure uses an array of edges **in_e** defining the block being clipped, and creates another array of edges **out_e** defining the clipped block. It uses procedure OUTPUT to place an edge into array **out_e**. The function LINE_INTERSECT calculates the intersection of the polygon edge from vertex s to vertex p with a clip boundary which is one of the edges defining the clip block. The function INSIDE returns **true** if a vertex lies inside (to the left) of the clip boundary. This test is based on the cross product of the vector P_1 to P_2 representing the clip edge, with the vector from P_1 to the vertex P_3. This cross product is a vector magnitude $x_{\underline{v}}y_{\underline{w}} - y_{\underline{v}}x_{\underline{w}}$ perpendicular to the plane defined by P_1, P_2 and P_3, where $\underline{v} = (x_{\underline{v}}, y_{\underline{v}})$ (vector from P_1 to P_2), and $\underline{w} = (x_{\underline{w}}, y_{\underline{w}})$ (vector from P_2 to P_3). The tested point is inside when $x_{\underline{v}}y_{\underline{w}} - y_{\underline{v}}x_{\underline{w}}$ has a negative value, otherwise the tested point is not inside the clip boundary.

3.2.3 Storing polygons in a black node of the reference OAQ

As discussed in the previous section, the node in reference OAQ (A) has two additional fields which are labelled **POLY_COUNT** and **POLY_LIST**. The first field is used to store the number of polygons defining all pairs of intersecting black blocks. The second field is used to store the polygon information about two intersecting black blocks from two different OAQs (i.e the output from ARBOR_BLOCK_INT). This field is an array of record structure termed **POLYGON**. The Pascal structure for

```
procedure ARBOR_BLOCK_INT(level_A,level_B : integer; var polygon : edge_ptr);

var i,j,k,num_int,in_length,out_length : integer; first_in,first_out : boolean;
    in_e,out_v : array[1..10] of edge; int,p,s,keep1,keep2 : vertex;
begin
    for i := 1 to 4 do begin
        with in_e[i] do begin
            vert1 := corners_B[i]; if i = 4 then vert2 := corners_B[1] else vert2 := corners_B[i+1];
            case i of 1 : edge_ID := 'BS'; 2 : edge_ID := 'BE'; 3 : edge_ID := 'BN'; 4 : edge_ID := 'BW'; end;
    end; { for i:= 1 to 4 }
    in_length := 4;
    for i := 1 to 4 do begin
        if i = 4 then m := 1 else m := i + 1;
        num_int := k_out := out_length := 0; first_in := first_out := false;
        for j := 1 to in_length do begin
            s := in_e[j].vert1; p := in_e[j].vert2;
            if INSIDE(p,corners_A[i], corners_A[m]) then
                if INSIDE(s,corners_A[i],corners_A[m]) then
                    OUTPUT(level_A,level_B,s,p,out_length, out_v,in_e[j].edge_ID)
                else begin { going in }
                    LINE_INTERSECT(s,p,corners_A[i], corners_A[m],int,found,1);
                    OUTPUT(level_A,level_B,int,p, out_length,out_v, in_e[j].edge_ID);
                    keep1 := int; { keep first intersection point }
                    num_int := num_int + 1; k_out := out_length; if not first_out then first_in := true; end
                else { going out }
                    if INSIDE(s,corners_A[i], corners_A[m]) then begin
                        LINE_INTERSECT (s,p,corners_A[i], corners_A[m],int);
                        OUTPUT(level_A,level_B,s,int, out_length,out_v,in_e[j].edge_ID);
                        keep2 := int; { keep second intersection point }
                        num_int := num_int + 1; if not first_in then first_out := true; end;
                    if num_int = 2 then begin { form keep1 keep2 as an edge }
                    case i of
                        1 : if keep2.x > keep1.x then {south boundary of block A }
                        OUTPUT(level_A,level_B,keep1,keep2, out_length,out_v,'AS'); else
                        OUTPUT(level_A,level_B,keep2, keep1,out_length,out_v,'AS');
                        2 : if keep2.y > keep1.y then { east boundary of block A }
                        OUTPUT(level_A,level_B,keep2, keep1,out_length,out_v,'AE') else
                        OUTPUT(level_A,level_B,keep1,keep2, out_length,out_v,'AE');
                        3 : if keep2.x > keep1.x then { north boundary of block A }
                        OUTPUT(level_A,level_B,keep2, keep1,out_length,out_v,'AN') else
                        OUTPUT(level_A,level_B,keep1,keep2, out_length,out_v,'AN');
                        4 : if keep2.y > keep1.y then { west boundary of block A }
                        OUTPUT(level_A,level_B,keep1, keep2,out_length,out_v,'AW') else
                        OUTPUT(level_A,level_B,keep2, keep1,out_length,out_v,'AW');
                    end; { case i of }
                    num_int := 0; { initialize the number of intersection points }
                    end; { if num_int = 2 }
        end; { for j := 1 to in_length }
        if first_out then swap edge kout with edge kout+1
        in_e := out_v; in_length := out_length;
    end; { for i := 1 to 4 }
    polygon ← out_v;
end
```

Figure 3.7: The ARBOR_BLOCK_INT algorithm for extracting a polygon produced by intersecting between two specified blocks.

the invariant part of a record **OAQ** is given in Figure 2.14 and Figure 3.2.

Consider Figure 3.8. This figure depicts an intersection between two OAQs along with their OAQ representation and their block decompositions. Each of the nodes is labelled 1,2,3,4 in order NW, NE, SW, SE respectively, and the root node is labelled 0.

Consider a black leaf node of the NW subtree of OAQ A (node 4 at level 2). Its block represented by the node with specified path 014 intersects with blocks represented by black leaf nodes of the SW subtree B of OAQ B level 2 and 1. These black leaf nodes have specified paths from the root node of B such as 0314, 0332, 0334 (at level 1) and 032, 034 (at level 2). Thus, they result in a set of polygons labelled in Figure 3.8 as P, R, V, Q, and T respectively. The intersection of the block of OAQ A and the blocks of OAQ B is shown in Figure 3.9. The polygon R is symbolically represented as shown in Figure 3.10. The information about each edge contains the level number of the node where the polygon came from. This is necessary when merging polygons to the parent node. The details of merging polygons is discussed in section 3.3. After storing all polygons defining all intersections between two blocks from different OAQs, OAQ B can be ignored, and the merging operation is carried out in OAQ A using polygon information stored at every black node of OAQ A. The algorithm for merging polygons follows the OAQ data structure by making use of edge information shown in Figure 3.10. The process of extracting polygons for Boolean operations with two OAQs can be summarized as shown in Figure 3.11.

3.3 Merging operations

This merging process is intended to merge all polygons which are produced by intersecting pairs of blocks represented by black nodes of different OAQs.

This process is done recursively by traversing the reference OAQ A in post order. For the intersection operation, when a process of traversal visits a BLACK node,

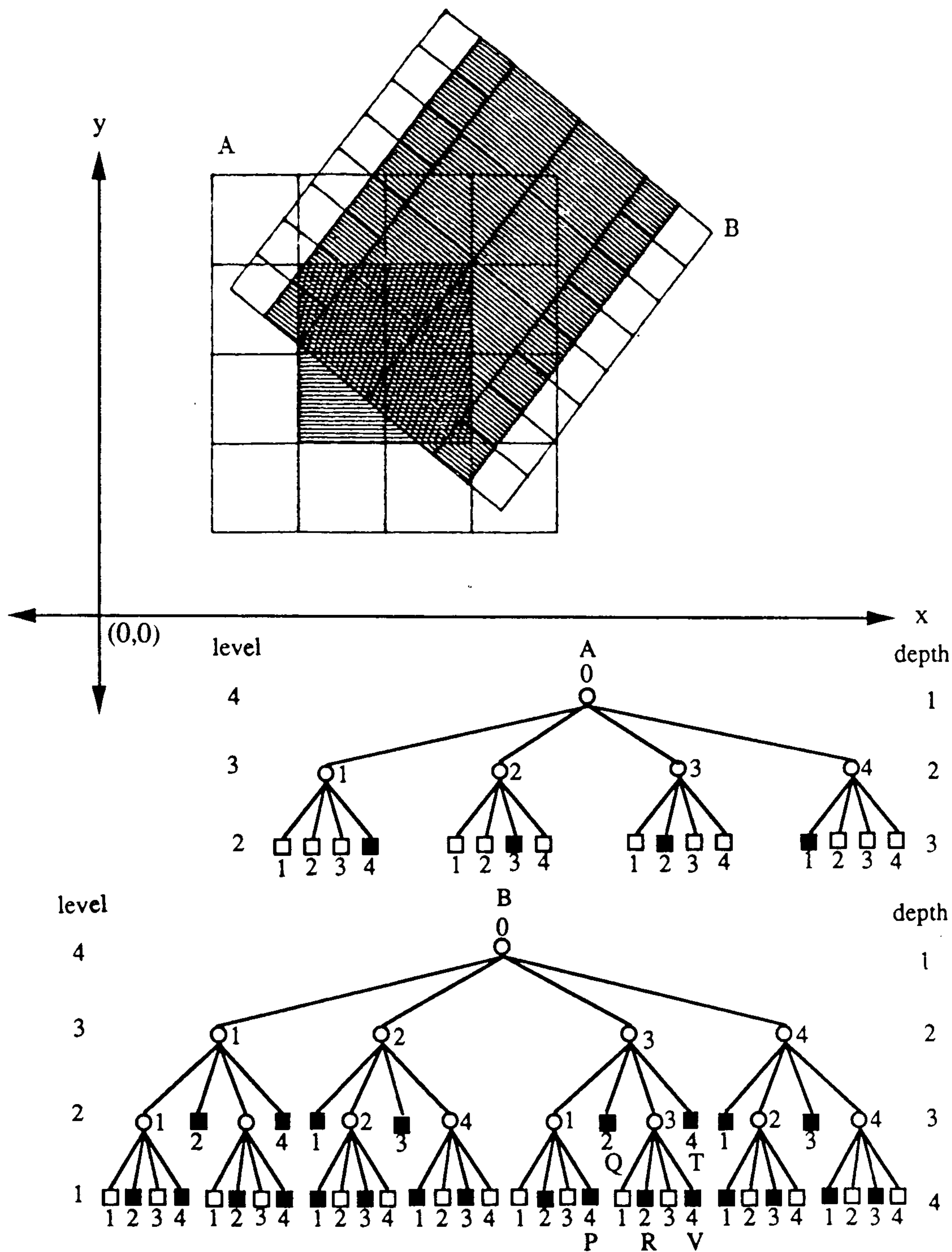


Figure 3.8: Intersection of two OAQs.

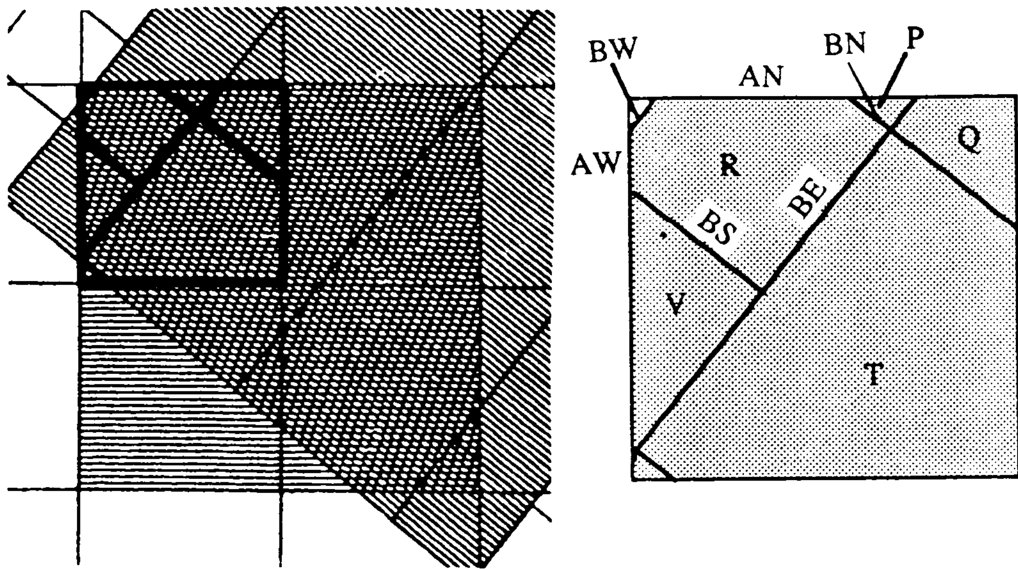


Figure 3.9: The intersection of the block of OAQ A and the blocks of OAQ B.

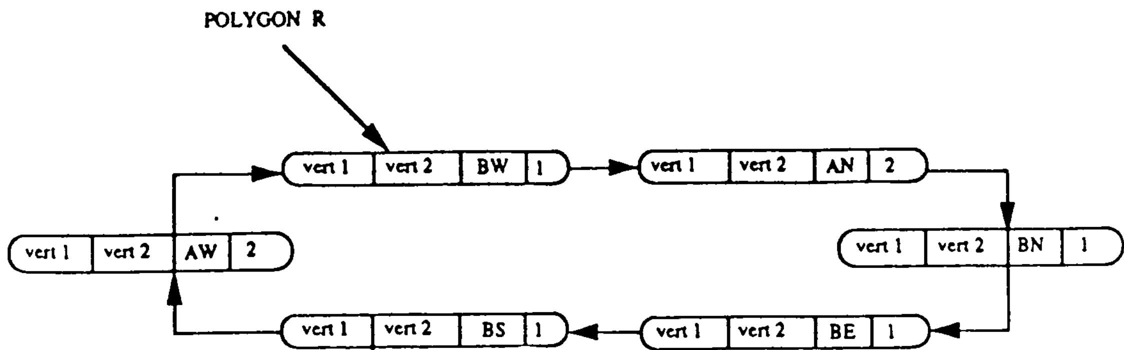


Figure 3.10: A set of edges representing polygon R from 3.9.

Intersection operation

1. Record each block of A and of B in the reference frame of A.
2. If node A and B are BLACK nodes then
   { Do a process of examining nodes }
   if block A and B intersect then
   Extract polygon intersection by invoking the ARBOR_BLOCK_INT procedure.
   else if block A and block B are disjoint no extraction polygon is carried out.
   else if block A and block B in the same position then polygon ← block A or block B.
   else if block A is inside block B then polygon ← block A
   else if block B is inside block A then polygon ← block B.
   store polygon in black node A.
3. If node A is BLACK and node B is GRAY then apply the process of examining nodes to the children of node B.
4. If node A is GRAY and node B is BLACK then apply the process of examining nodes to the children of node A.
5. If both nodes A and B are GRAY then
   if the root nodes are disjoint then no further operation is carried out.
   else apply the process of examining nodes recursively (step 2 above)
   to the children of node A and node B.

Union operation

1. Record each block of A and of B in the reference frame of A.
2. If node A is BLACK then polygon ← block A.
3. If node A is WHITE and node B is BLACK then
   if block A and B intersect then
   Extract polygon intersection by invoking the ARBOR_BLOCK_INT procedure.
   else if block A and block B are disjoint no polygon extraction is carried out.
   else if block A and block B in the same position then polygon ← block A or block B.
   else if block A is inside block B then polygon ← block A
   else if block B is inside block A then polygon ← block B.
   store polygon in black node A.
4. If node A is WHITE and node B is GRAY then apply the process of examining nodes to the children of node B.
5. If node A is GRAY and node B is WHITE then apply the process of examining nodes to the children of node A.
6. If both nodes A and B are GRAY then
   if the root nodes are disjoint then no further operation is carried out.
   else apply the process of examining nodes recursively (step 2 above)
   to the children of node A and node B.

Figure 3.11: Steps of extracting polygons for Boolean operations with two OAQs.

the stored polygons are merged, which results in another polygon which is stored in the same node. For the union operation, if a process of traversal visits either a BLACK or a WHITE node, the same action is taken. If a traversal visits a GRAY node, the polygons which are stored in children nodes are merged. For intersection, the polygons to be merged are stored in the BLACK children nodes. The resulting polygons are stored in the GRAY node. Finally, after visiting all children nodes, the traversal visits the root node and merges all polygons which are stored in the children nodes. The final result is one or more polygons which can be represented into one or more OAQs.

This algorithm is depicted in Figure 3.12, and shows that if the traversal visits a BLACK node, the stored polygons are merged by calling procedure MERGE_OPERATION_B, which is depicted in Figure 3.13. If a traversal of OAQ A visits a GRAY node, then the algorithm merges all polygons stored in the node's children by invoking procedure MERGE_OPERATION_A. This procedure is depicted in Figure 3.14. The details of the MERGE_OPERATION_B, known as method I, is discussed in section 3.3.1. Another merging method, known as method II, is discussed in section 3.3.2.

3.3.1 Method I

This subsection details the process of merging polygons which are stored in a BLACK node of the reference OAQ A, without considering OAQ B anymore. The information of each polygon stored in the **HISTORY** and **DEPTH** fields are necessary to merge polygons and result in another polygon.

To clarify this algorithm, consider Figure 3.8. This figure depicts intersection between two OAQs. A block representing a black node of OAQ A with path 014 from the root intersects blocks representing nodes of OAQ B with paths 0314, 032, 0332, 0334, and 034 respectively. Thus **POLY_LIST** stores the result of those intersections as polygons with **HISTORY** and **DEPTH** in the order

```
depth_A := 0;
procedure MERGE_OPERATION(var head : pqtree);

var p,q : pqtree; q1 : edge_ptr; k : integer;
begin
p := head;
if p ≠ nil then begin
    depth_A := depth_A + 1;
    history_A[depth_A] := 1; MERGE_OPERATION(p^.NW);{ traverse to NW son }
    history_A[depth_A] := 2; MERGE_OPERATION(p^.NE); { traverse to NE son }
    history_A[depth_A] := 3; MERGE_OPERATION(p^.SW); { traverse to SW son }
    history_A[depth_A] := 4; MERGE_OPERATION(p^.SE);{ traverse to SE son }
    depth_A := depth_A - 1;
    if (p^.colour ≠ 'G') {'B' for intersection, 'W' for union } and (p^.poly_count > 0) then
    begin
        if p^.poly_count > 1 then begin { merge the stored polygons which are from OAQ B }
            MERGE_OPERATION_B (p^.poly_list,p^.poly_count);
            p^.poly_list[1].history := history_A;
            p^.poly_list[1].depth := depth_A; end;
    if (p^.colour = 'G') then begin
        for k := 1 to 4 do begin
            case k of 1 : q := p^.NW; 2 : q := p^.NE; 3 : q := p^.SW; 4 : q := p^.SE; end;
            if q^.poly_count > 0 then begin { if only one polygon found in the son nodes }
                p^.poly_count := p^.poly_count + 1;
                p^.poly_list[p^.poly_count] := q^.poly_list[1]; { store polygon in the GRAY node }
            end
        end;
    if p^.poly_count > 0 then { more then one polygons found }
        MERGE_OPERATION_A (p^.poly_list,p^.poly_count,'A'); { merge polygons }
end;
end; { of MERGE_OPERATION }
```

Figure 3.12: The MERGE_OPERATION algorithm for merging polygons for boolean OAQ operations.

```
procedure MERGE_OPERATION_B (poly_list : polygon_array; var poly_count : integer; )

var S_F : array [1..10,1..2] of integer;
    a_start : array [1..10] of integer;
    i,j,k,key,limit,maxS_F start,finish,parent,c : integer;
begin
limit := 0;
SORT_POLYGON_LIST(poly_list,poly_count);
repeat; c := 0;
    GROUP_POLYGON_LIST (poly_list,poly_count,S_F,maxS_F);
    for i := 1 to maxS_F-1 do begin
        SORT_PARENT(poly_list,S_F[i,1],S_F[i+1,1]-1);
        for parent := 0 to 4 do begin
            GET_SAME_PARENT_POLYGONS (poly_list,S_F[i,1],S_F[i+1,1]-1,parent, start,finish);
            if (finish ≠ 0) and (poly_count > 1) then begin { more than one polygon found }
                c := c + 1;  a_start[c] := start;
                if (finish - start) ≥ 1 then begin { merge polygons whose the same parent nodes }
                    MERGE_SAME_PARENT_POLYGONS (start,finish-start+1,poly_list,'B');
                    for j := start+1 to finish do begin
                        dispose(poly_list[j].edge_list);
                        poly_list[j].edge_list := nil;
                    end;
                end
                else begin { if only one polygon found }
                    key := poly_list[start].history[poly_list[start].depth];
                    UPDATE_POLYGON (key,poly_list[start].edge_list,'B', poly_list[start].depth)
                end;
                poly_list[start].depth := poly_list[start].depth - 1;
            end { if more than one polygons found }
        end; { for parent }
    end; { for i }
    for k := 1 to c do poly_list[k] := poly_list[a_start[k]]; { update polygon list }
    poly_count := c;
    limit := limit + 1;
until (poly_count = 1) or (limit = maxlevel)
end; { of procedure MERGE_OPERATION_B }
```

Figure 3.13: The MERGE_OPERATION_B algorithm for merging all polygons which are stored in the node of OAQ A; i.e. BLACK node for intersection operation, WHITE node for union operation.

```
procedure MERGE_OPERATION_A(var poly_list : polygon_array; var poly_count : integer);

var j,key : integer; q1 : edge_ptr;
begin
 if poly_count > 0 then begin { Check if there is/are polygon(s) to be merged/updated }
    for j := 1 to poly_count do begin { Update each polygon which is stored in each son node }
        key := poly_list[j].history[poly_list[j].depth];
        UPDATE_POLYGON (key,poly_list[j].edge_list,' A',poly_list[j].depth);end
 end;
 if poly_count > 1 then begin { merge all polygons which are stored in the son nodes }
    MERGE_SAME_PARENT_POLYGONS(1,poly_count, poly_list,'A');
    for j := 2 to poly_count do begin
        dispose(poly_list[j].edge_list);
        poly_list[j].edge_list := nil;
    end;
 end
end; { of procedure MERGE_OPERATION_A }
```

Figure 3.14: The MERGE_OPERATION_A algorithm for merging all polygons which are stored in GRAY children nodes.

HISTORY	DEPTH	POLYGON
0314	4	P
032	3	Q
0332	4	R
0334	4	V
034	3	T

HISTORY shows the path where the polygon came from , and **DEPTH** shows the number of nodes traversed on the path from the root node to this node. If the polygon belongs to the OAQ root, the **DEPTH** is 1. The idea of merging the stored polygons in black nodes of reference OAQ A is taken directly from the idea of traversing OAQ B in post order traversal. When the merge algorithm visits a GRAY node, it merges all polygons belonging to children of the node. The polygons to be merged are stored in a node of OAQ A as a list of polygons, and OAQ B is no longer considered.

Figure 3.8 shows that the polygons labelled P, R, and V belong to nodes of OAQ B with paths from the root 0314, 0332, and 0334. These three nodes are in the first

level of OAQ B. The polygons labelled Q and T belong to the nodes of OAQ B with paths from the root 032 and 034. They are in the second level. The information about which polygons in the list belong to nodes of OAQ B with the same level can be taken from the **DEPTH** field. The information about which polygons in the list belong to nodes of OAQ B which have the same father in the same level, can be taken from the **HISTORY** field.

From the list of polygons, we can group all polygons which belong to nodes in OAQ B with the same level, by sorting the list using **DEPTH** as the principal sort field. In our example, the list of polygons which is stored in a black node of OAQ A consists of two groups. The first group consists of polygons P, R and V. The second group has polygons Q and T. Each group is now classified to get polygons which belong to nodes of OAQ B with the same parent. This is done by sorting the group using **DEPTH − 1** as the sort key. This allows merging of polygons which belong to nodes with the same parent. As an example, the first group of the list of polygons stored in the black node of OAQ A with path 014, consists of polygons P, R and V belonging to nodes of OAQ B with paths 0314, 0332, 0334. These polygons are classified into two groups by sorting the **HISTORY** field using its third element as the principal sort key, since the **DEPTH** field has the value 4. The result is a group consisting of polygon P, and another group consisting of polygons Q and V. Polygons which belong to the same parent node are merged, and the corresponding **HISTORY** is updated to be the **HISTORY** of the parent node by taking off the last **HISTORY** element and decrementing **DEPTH** by 1. The information belonging to polygon P is updated to the higher level, since there is no polygon with the same parent node to be merged. Updating the information is explained in subsection 3.3.3. The updated polygon P now belongs to the node with path 031, and a merged polygon R and V belongs to node 033. These two polygons are merged and stored in the first element of the list of polygons.

After finishing this process, the corresponding result is a polygon which is stored

in the node of OAQ A. **HISTORY** and **DEPTH** are changed to reflect the structure of OAQ A. Then, OAQ A is traversed in post order. If a traversal visits a GRAY node then merge all polygons stored in the node's children . The method to merge the polygons is similar to that discussed above.

3.3.1.1 Merging polygons belonging to nodes with the same parent

A group of polygons which belong to nodes of an OAQ with the same parent, can now be merged. The information about which edges are coincident, can be found by making use of the **DEPTH** and **HISTORY** fields. For example, polygon R and V (see Figure 3.8) belong to nodes of OAQ B which have the same parent. The fourth element of **HISTORY** of polygon R is **HISTORY**[4] = 2 and for polygon V **HISTORY** [4] = 4. Knowing these numbers, we know that these polygons belong to the NE node and the SE node without knowing exactly the picture of OAQ B. Figure 3.15 shows a subtree of OAQ B and its children as well as the block of OAQ A

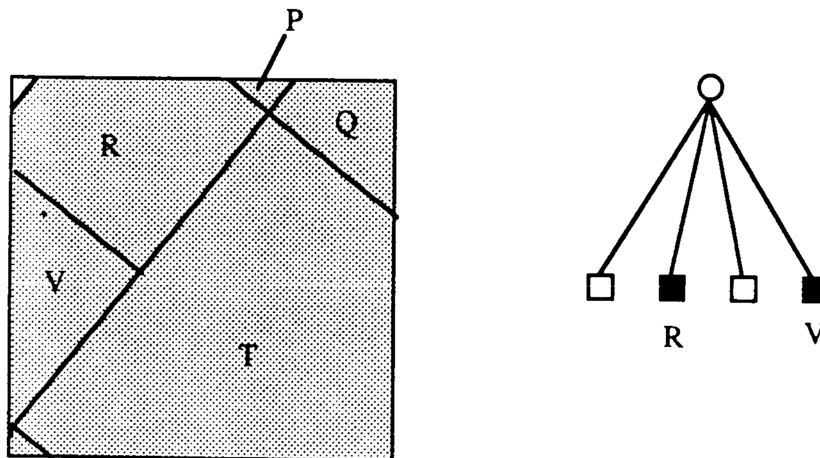


Figure 3.15: Subtree of OAQ B with its polygons.

containing polygons R and V. A 'BS' edge of polygon R coincides with a 'BN' edge of polygon V. To get the resulting merged polygon, polygon R must be linked to polygon V by removing the 'BS' edge of polygon R and the 'BN' of polygon V. A similar manner is taken when merging polygons belonging to the same parent.

There are nine possible combinations of types of polygons with the same parent node being merged. If the polygons being merged belong to an OAQ's node which is labelled 1 and 2 or 3 and 4, say polygon types 1 and 2 or 3 and 4, the possible edges to be removed for merging them are an 'E' (east) edge for polygon types 1 and 3, and 'W' (west) edge for polygon types 2 and 4.

If polygons to be merged are types 1 and 3 or 2 and 4, then the possible edges to be removed are 'S' (south) edge for polygon types 1 and 2, and 'N' (north) edge for polygon type 3 and 4.

If polygon types 1, 2 and 3 are being merged, then the corresponding edges to be removed are 'S' and 'E' edges for polygon type 1, and 'W' edge for polygon type 2, and 'N' edge for polygon type 3. If polygons types 1, 2 and 4 are being merged, then the possible edges to be removed are 'E' for polygon 1, 'S' and 'W' for polygon type 2, and 'N' for polygon type 4. If polygons to be merged are of types 2,3 and 4 then the possible edges to be removed are 'S' edge for polygon type 2, 'E' edge for polygon type 3, and 'N' and 'W' edges for polygon type 4.

If polygons types 1,2,3 and 4 are being merged then the possible removed edges are 'S' and 'E' for polygon type 1, 'W' and 'S' for polygon type 2, 'N' and 'E' for polygons type 3, and 'W' and 'N' for polygon type 4.

The type of each polygon is known directly from the last element of its **HISTORY** which is known as **HISTORY[DEPTH]**. The possible combinations of polygon types to be merged are depicted in Figure 3.16.

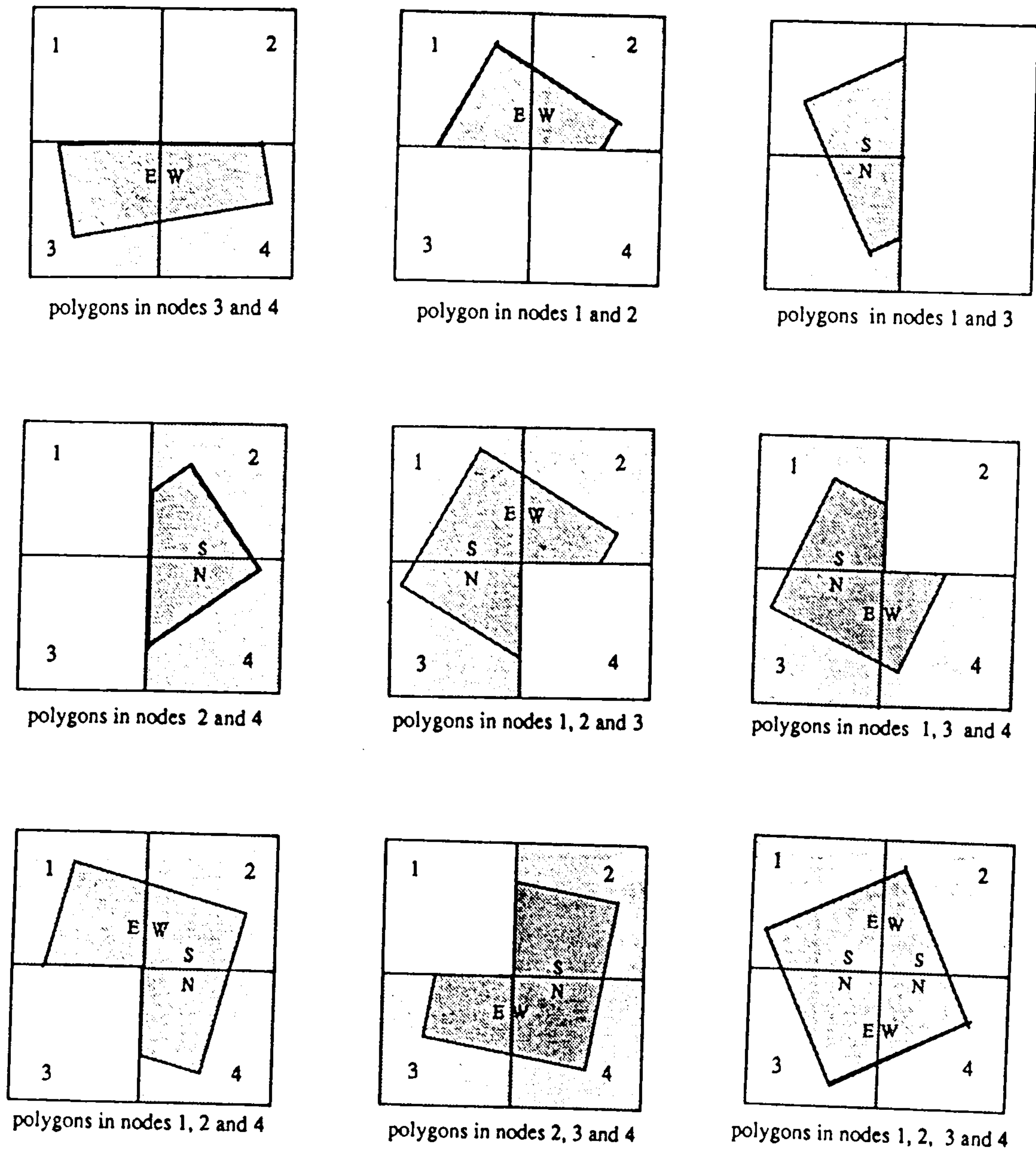


Figure 3.16: Possible combinations of polygon types being merged.

3.3.2 Method II

In general, a process of merging a set of polygons resulting from operations between two OAQs requires a traversal of the reference OAQ A in post order. When traversal visits the non-GRAY node (BLACK node for the intersection operation and BLACK or WHITE node for the union operation) which has a collection of polygons, the polygons are merged and stored in the same node. When the traversal visits a GRAY node, then the polygons which are stored in the children of the GRAY node are merged. The root of the reference OAQ A stores the final result.

For example, consider Figure 3.8, where a traversal of OAQ A visits a node with path 014, and the polygons stored in the node are merged. The first method makes use of the **DEPTH** field as principal sort key to classify the polygons into sets of polygons which belong to the same level node of OAQ B. Knowing the set of polygons which belong to the same level nodes of OAQ B, this set is classified into groups of polygons which belong to the nodes of OAQ B with the same parent. This is done by sorting the set of polygons using **HISTORY[DEPTH - 1]** as the principal sort key. Knowing the class of polygons which belong to the same parent node of OAQ B, the polygons are merged. The method of merging polygons by sorting the set of polygons has been discussed in subsection 3.3.1.

The second method does not require any sorting. Instead, a temporary subtree of OAQ B which contributes to the collection of polygons is created. For example, when a traversal of OAQ A visits a BLACK node, then the polygons P,Q,R,V and T stored in the node are merged. Using the **HISTORY** and **DEPTH** information, a temporary subtree of OAQ B is made (see Figure 3.17). After creating the temporary subtree, it is traversed in post order. When a traversal visits a GRAY node of subtree A, then the process merges the polygons which are stored in the children nodes. If only one polygon is found then the polygon is updated to the GRAY node. If more than one polygon is found in the children of the GRAY node of the subtree, then they are merged using the method of merging polygons with the same parent nodes,

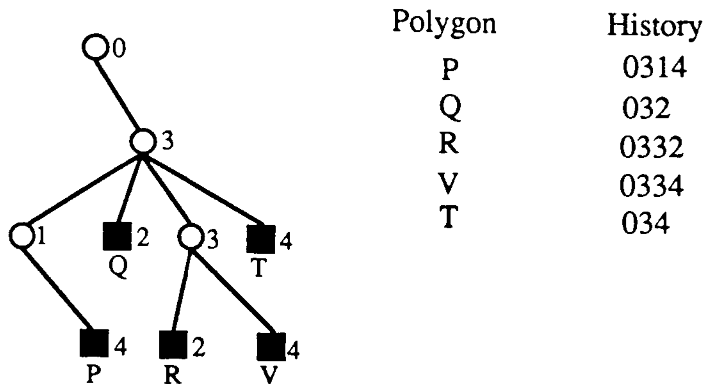


Figure 3.17: The temporary subtree of OAQ B.

which has been discussed in section 3.3.1.1. The algorithms for creating a temporary subtree of OAQ B, and for merging polygons which belong to the subtree are shown in Figures 3.18 and 3.19.

3.3.3 Updating polygons

The process of updating polygons is needed in case only one polygon is found in the node. Polygon P belonging to the node of path 0314 of OAQ B, which as shown in Figure 3.8, cannot be merged with any other polygons belonging to the same parent node. Thus, this polygon is updated to be a polygon belonging to its parent node (031).

As shown in Figure 3.15, polygon P has an edge list with edge identifications, AN, BS, and BE, respectively. Its edge levels are 2, 1 and 1 respectively. Once this polygon is updated, it is owned by the node 031 which is the parent node of node 0314. The BS's and BE's edge levels are assigned the value of the parent node's level. The type of polygon to be updated is known directly from the last element of

```
procedure CREATE_SUBTREE(var r : pqtree; poly_list : polygon_array; var count : integer);

var root,p,q : pqtree; k,n,depthA : integer; historyA : abyte;
begin
   new(root); root^.NW := nil; root^.NE := nil; root^.SW := nil; root^.SE := nil;
   root^.colour := 'G'; root^.count := 0;
   for k := 1 to count do begin
      p := root; q := root;
      for n := 2 to poly_list[k].depth do begin { create subtree }
         case poly_list[k].history[n] of
            1 : begin q := p; p := p^.NW; end; 2 : begin q := p; p := p^.NE; end;
            3 : begin q := p; p := p^.SW; end; 4 : begin q := p; p := p^.SE; end; end;
         if p = nil then begin
            new(p); { allocate a new node }
            p^.NW := nil; p^.NE := nil; p^.SW := nil; p^.SE := nil;
            p^.colour := 'G'; p^.count := 0; end;
         case poly_list[k].history[n] of { link the node to it's parent }
            1 : q^.NW := p; 2 : q^.NE := p; 3 : q^.SW := p; 4 : q^.SE := p; end;
      end;
      { at leaf node store the kth polygon }
      p^.poly_list[1] := poly_list[k]; p^.count := 1; p^.colour := 'B';
   end; { for k = 1 }
   depth_B := 1; history_B[depth_B] := 0;
   MERGE_SUBTREE(root); { merge all polygons stored in subtree }
   r^.count := root^.count; for k := 1 to root^.count do
      r^.poly_list[k] := root^.poly_list[k]; { result of merging }
end; { of procedure CREATE_SUBTREE }
```

Figure 3.18: The CREATE_SUBTREE algorithm for creating a temporary subtree of OAQ B.

```
procedure MERGE_SUBTREE(var head : pqtree);

var p,q,r : pqtree; k,i,key : integer;
begin
    p := head;
    if p ≠ nil then begin
        depth_B := depth_B + 1;
        history_B[depth_B] := 1; MERGE_SUBTREE(p^.NW); { traverse to the NW child }
        history_B[depth_B] := 2; MERGE_SUBTREE(p^.NE); {traverse to the NE child }
        history_B[depth_B] := 3; MERGE_SUBTREE(p^.SW); { traverse to the SW child }
        history_B[depth_B] := 4; MERGE_SUBTREE(p^.SE); { traverse to the SE child }
        depth_B := depth_B - 1;
        if p^.colour = 'G' then begin
            p^.count := 0;
            for k := 1 to 4 do begin { copy the stored polygons from it's children }
                case k of 1 : q := p^.NW; 2 : q := p^.NE; 3 : q := p^.SW; 4 : q := p^.SE; end;
                if q ≠ nil then for i := 1 to q^.count do begin
                    p^.count := p^.count + 1; p^.poly_list[p^.count] := q^.poly_list[1]; end;
            end; { of copying }
            if p^.count > 1 then begin { merge polygons in the GRAY node of subtree }
                new(r); r^.count := 0;
                MERGE_OPERATION_A(r,p^.poly_list, p^.count,'B');
                for k := 1 to r^.count do begin
                    p^.poly_list[k] := r^.poly_list[k]; p^.count := r^.count; end
            end
            else if p^.count = 1 then begin { only one polygon found }
                key := p^.poly_list[1].history[p^.poly_list.depth]; { update polygon }
                UPDATE_POLYGON(key,p^.poly_list[1],'B', p^.poly_list[1].depth);
                p^.poly_list[1].depth := p^.poly_list[1].depth - 1; end;
        end; { if colour = 'G' }
    end;
end; { of procedure MERGE_SUBTREE }
```

Figure 3.19: The MERGE_SUBTREE algorithm for merging polygons which belong to the subtree of OAQ B.

HISTORY, that is **HISTORY[DEPTH]**.

Recall that each GRAY node of an OAQ's subtree has 4 blocks representing the children nodes (see Figure 3.20). The blocks have outer boundaries labelled N,W edges for NW block, N,E edges for NE block, S,E edges for SE block, and S,W edges for the SW block. For a polygon to be updatable, the following criteria must be

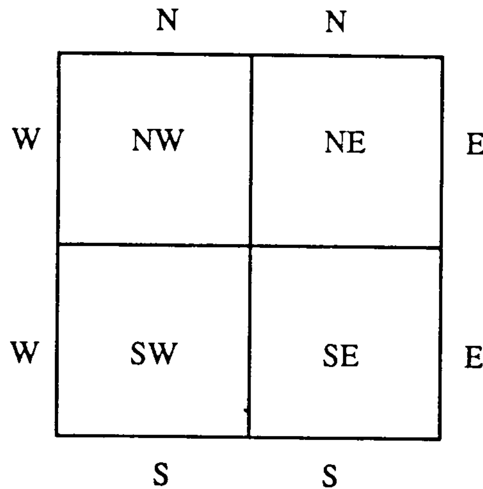


Figure 3.20: Blocks representing a GRAY node.

fulfilled.

If the polygon to be updated belongs to NW node, then the edge level of N,W edges must be updated by assigning them the value of the level of the parent node. For a polygon which belongs to a NE node, the updated edges are those labelled N,E. For a polygon which belongs to SW node the updated edges are S,W. For a polygon which belongs to a SE node, the updated edges are S, E.

The process of updating polygons is repeated until other polygons belonging to nodes in the same level are found or until the maximum level of the OAQ is reached. Consider a subtree of OAQ A along with the merged polygons represented by blocks P,Q, and R shown in Figure 3.21.

Polygons P and Q belonging to nodes 0322 and 0323, cannot be merged directly

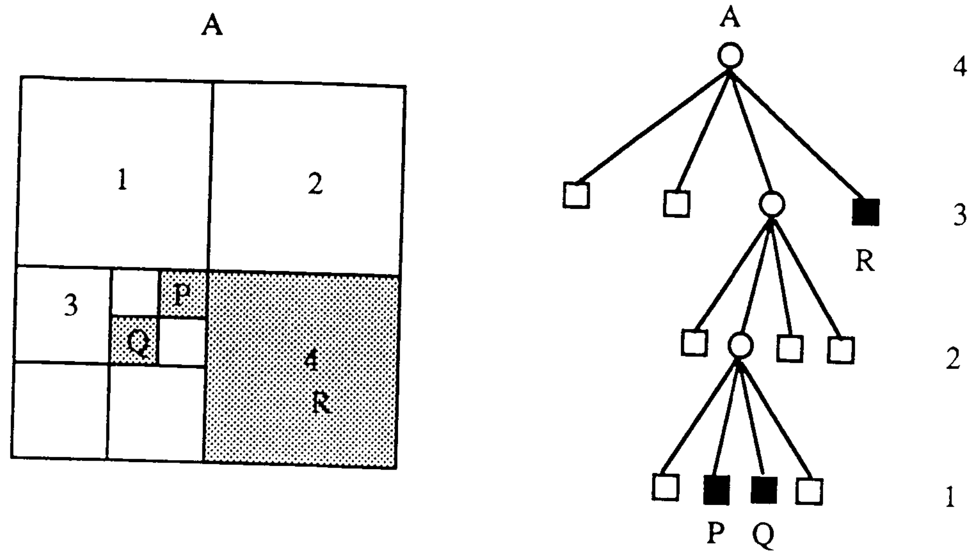


Figure 3.21: The subtree of an OAQ illustrating polygon updating.

with the SE block R in level 3, which belongs to a node 04. Polygons P and Q must be updated twice, resulting in them being owned by node 03. Polygon P in the figure has **HISTORY** values 0322 and polygon Q has **HISTORY** values 0323.

Polygon P, which belongs to node 0322, has successive edge identifications and their edge levels represented as a list of (BW, 1), (BS, 1), (BE, 1) and (BN, 1). The **HISTORY** of polygon P has a value of 0322, and the **DEPTH** value is 4. The last **DEPTH** element of **HISTORY** gives the type of the polygon (2 or NE). To update this polygon to be a polygon belonging to its parent node, the N and E edges must be updated by assigning the level of the parent node to the edge level of N and E edges. This process is repeated level by level, until the polygon can be merged with other polygons belonging to a node in the same level. In our example, polygon P is updated twice, and the steps of updating the polygon are shown in Figure 3.22. The same process is taken to update polygon Q, and this is also shown in Figure 3.22. It should be noticed that the second process of updating polygon Q changes the AE and AN edge levels to be 3. Since the difference of edge levels before and after updating is greater than 1, the polygon cannot be updated. Furthermore, the polygon cannot

Polygon P									
HISTORY	ID	level	ID	level	ID	level	ID	level	updated
0322	AW	1	AS	1	AE	1	AN	1	
						↓		↓	
032	AW	1	AS	1	AE	2	AN	2	1st
						↓		↓	
03	AW	1	AS	1	AE	3	AN	3	2nd

Polygon Q									
HISTORY	ID	level	ID	level	ID	level	ID	level	updated
0323	AW	1	AS	1	AE	1	AN	1	
		↓		↓					
032	AW	2	AS	2	AE	1	AN	1	1st
						↓		↓	
03	AW	2	AS	2	AE	3	AN	3	2nd

Figure 3.22: Process of updating polygon P and Q.

be merged with any other polygons which belong to nodes in the same level (see Figure 3.21). The information from the previous updating is kept until the process of merging polygons reaches the OAQ root.

The algorithm for updating polygons shown in Figure 3.23.

3.4 Intersection operation between two OAQs

In discrete mathematics, intersection is one of the operations on sets, which is defined as [33]

$$A \bigcap B = \{x | x \epsilon A \ \ and \ \ x \epsilon B\} \tag{3.8}$$

The Venn diagram is shown in Figure 3.4, the interior of the large rectangle represents the universal set, U, while the shading is used to represent particular sets. The Venn diagram of the intersection of A and B is shown as the darker shaded portion.

The same concept is used when computing the intersection between two OAQs.

```
procedure UPDATE_POLYGON(key : integer; var poly : polygon; c : char; depth : integer);

var update_edge : packed array[1..2] of char; i,newlevel : integer;
p : edge_ptr; found1,found2 : boolean;
begin
   found1 := false; found2 := false;
   case key of 1 : update_edge := 'NW'; 2 : update_edge := 'NE';
   3 : update_edge := 'SW'; 4 : update_edge := 'SE'; end;
   newlevel := maxlevel - depth + 2; {update edge level }
   p := poly.edge_list; i := 0;
   repeat
   i := i + 1;
   if(p^.edge_ID[1] = c) and
      ((p^.edge_ID[2] = update_edge[1]) or (p^.edge_ID[2] = update_edge[2])) then begin
      if(newlevel - p^.level) = 1 then
      p^.level := newlevel; { assign updated level }
         if not found1 then found1 := true else found2 := true; end;
      p := p^.next
   until (p = poly.edge_list) or (found2);
end; { of procedure UPDATE_POLYGON }
```

Figure 3.23: The UPDATE_POLYGON algorithm for updating a polygon to be a polygon owned by its parent node.

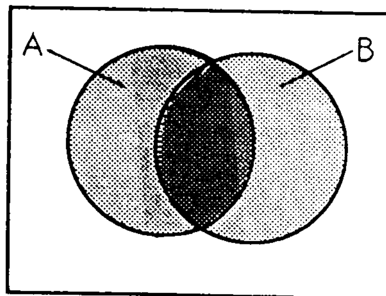


Figure 3.24: The Venn diagram of the intersection of A and B (from [33]).

Knowing the OAQ representations of two objects, the intersection operation between two OAQs can be carried out. The first step is to traverse both OAQ A and B in parallel while examining the nodes of OAQ A and OAQ B, following the steps outlined in Figure 3.11. This process is intended to extract all pairs of intersections of black blocks with each pair represented by two BLACK nodes, since there is no intersection between two blocks representing WHITE and BLACK nodes. If each pair of BLACK nodes of OAQ A and of OAQ B represent the intersection of two blocks, then the resulting polygon is extracted. The method of extracting polygons of two intersecting blocks has been discussed in subsection 3.2.2. If the blocks are disjoint then there is no extraction polygon. If the one of the blocks lies inside the other then the resulting polygon is the one which is inside the other. If the blocks coincide, then one of them is the resulting polygon.

The algorithm for examining all pairs of BLACK nodes of OAQ A and OAQ B is shown in Figure 3.25. Procedure RECORD_CORNERS records four corner points of a block of OAQ B in the OAQ A reference frame. The method of recording has been discussed in subsection 3.2.1. Function A_B_INTERSECT is used to check whether a block of OAQ A and a block of OAQ B intersect or not. The test is based on the line segment intersection test. Each edge of block of OAQ B is tested against four successive edges of block of OAQ A. If one of the edges of block B and one of the edges of block A intersect then the function returns true, otherwise a check is repeated until all the edges of the block of OAQ B have been tested. The test for line intersection is based on the parametric method. This technique is one of the most effective for determining whether two line segments intersect or for detecting singularities (colinearity, parallelism) [10]. Consider Figure 3.26. The figure depicts the intersection between two line segments.

The position of point (x, y) on the straight line is indicated by the value of the parameter. The line segment to which they belong are defined parametrically [10].

```
procedure INT_CHECK_NODES(q1 : pqtree; q2 : pqtree; level_A,x1,y1, level_B,x2,y2 : integer);

var i,k,num_edge : integer; polygon : edge_ptr;
block_intersect,found, disjoint,same_block,A_in_B,B_in_A : boolean;
begin
   found := disjoint := same_block := A_in_B := B_in_A := false;
   RECORD_CORNERS(level_A,x1,y1,level_B,x2,y2);
   if (q1^.colour = 'B') and (q2^.colour = 'B') then begin
      depth_A := depth_A + 1; depth_B := depth_B + 1; new(polygon);
      if A_B_INTERSECT then begin
      {block of OAQ A and block of OAQ B intersect }
         ARBOR_BLOCK_INT(level_A,level_B,polygon,num_edge); found := true; end
      else begin {quad A=quad B or quad A inside B or quad B inside A }
         CHECK_BLOCKS(disjoint,same_block,A_in_B,B_in_A, level_A,level_B);
         if disjoint then found := false
         else begin EXTR_BLOCK(level_A,level_B,same_block, A_in_B,B_in_A,polygon);
            found := true; end;
      end;
      if found then begin { store a polygon in the node of OAQ A }
         with q1^ do begin
            poly_count:= poly_count+ 1; POLY_LIST[poly_count].edge_list := polygon;
            POLY_LIST[poly_count].depth := depth_B;
            POLY_LIST[poly_count].history := history_B; end;
   end; end;
   if (q1^.colour = 'B') and (q2^.colour = 'G')
      for the four children of q2 do call INT_CHECK_NODES;
   if (q1^.colour = 'G') and (q2^.colour = 'B');
      for the four children of q1 do call INT_CHECK_NODES;
   if (q1^.colour = 'G') and (q2^.colour = 'G') then begin
      level_A := level_A - 1; level_B := level_B - 1; block_intersect := false;
      if A_B_INTERSECT then block_intersect := true;
      else  CHECK_BLOCKS(disjoint,same_block,A_in_B, B_in_A,level_A,level_B);
      if  block_intersect or same_block or A_in_B or B_in_A then begin
         depth_A := depth_A + 1; depth_B := depth_B + 1;
         for the four children of q1 do
            for the four children of do call INT_CHECK_NODES; end
   end
   else if (level_A = maxlevel_A) and (level_B = maxlevel_B) then
      writeln ('Images are completely disjoint ');
end; { of procedure INT_CHECK_NODES }
```

Figure 3.25: The INT_CHECK_NODES algorithm for examining nodes of OAQ A and B for intersection Boolean operation.

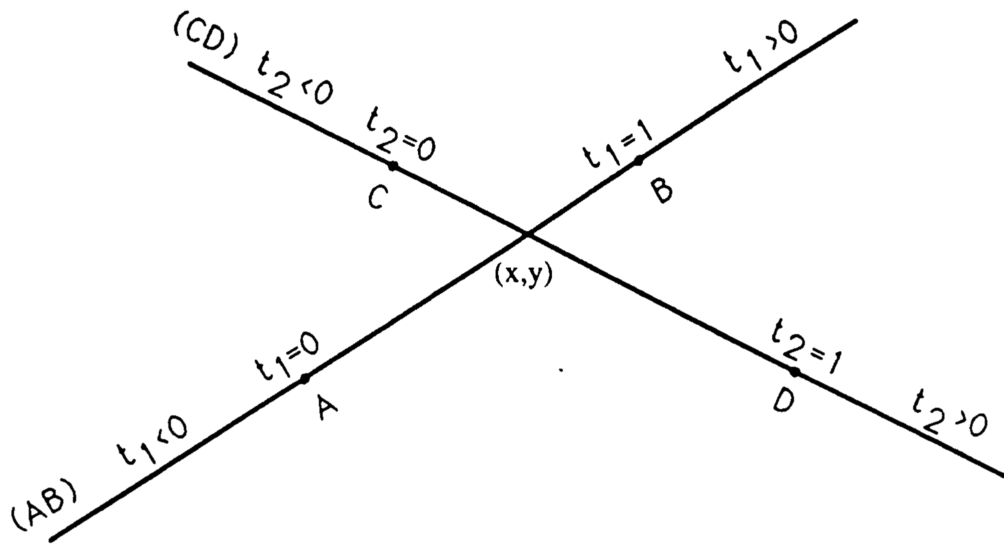


Figure 3.26: Intersection of two line segments.

For the line segment AB the parametric equation is defined as

$$\begin{aligned} x &= x_A + (x_B - x_A)t_1 \\ y &= y_A + (y_B - y_A)t_1 \end{aligned} \tag{3.9}$$

where (x_A, y_A) is the first endpoint of the line segment AB, and (x_B, y_B) is the second endpoint of the line segment AB. For the line segment CD the parametric equation is defined as

$$\begin{aligned} x &= x_C + (x_D - x_C)t_2 \\ y &= y_C + (y_D - y_C)t_2 \end{aligned} \tag{3.10}$$

where (x_C, y_C) is the first endpoint of the line segment CD, and (x_D, y_D) is the second endpoint of the line segment CD.

If the line segments AB and CD intersect, then the coordinates of the intersection point (x, y) are defined as

$$\begin{aligned} x &= x_A + (x_B - x_A)t_1 \\ &= x_C + (x_D - x_C)t_2 \end{aligned} \tag{3.11}$$

$$\begin{aligned} y &= y_A + (y_B - y_A)t_1 \\ &= y_C + (y_D - y_C)t_2 \end{aligned} \tag{3.12}$$

The solution of the above equations is :

$$\begin{aligned} t_1 &= \frac{(x_C - x_A)(y_C - y_D) - (x_C - x_D)(yC - y_A)}{(x_B - x_A)(y_C - y_D) - (x_C - x_D)(yB - y_A)} \\ t_2 &= \frac{(x_B - x_A)(y_C - y_A) - (x_C - x_A)(xB - y_A)}{(x_B - x_A)(y_C - y_D) - (x_C - x_D)(yB - y_A)} \end{aligned} \tag{3.13}$$

For determining colinearity or parallelism of two line segments, the algorithm makes use of denominator's value of equation (3.13). If this value is zero then the line is parallel, or colinear; otherwise one of the following conditions is fulfilled. If

$t_1 < 0$ or $t_1 > 1$ or $t_2 < 0$ or $t_2 > 1$ then the line segments do not intersect. If $t_1 = 0$ then the intersection point is at point A, the first endpoint of the line segment AB. If $t_1 = 1$ then the intersection point is at point B, the second endpoint of the line segment AB. If $t_2 = 0$ then the intersection point is at point C, the first endpoint of the line segment CD. If $t_2 = 1$ then the intersection point is at point D, the second endpoint of the line segment CD. If none of the above conditions are fulfilled then the line segments do cross, with the intersection point (x, y) whose coordinates are computed from equations (3.11) and (3.12).

The above method is used to check if the block of OAQ A and the block of OAQ B intersect. Once the intersection of two edges from a block of OAQ A and a block of OAQ B is found, then the function A_B_INTERSECT returns true. If a block of OAQ A and a block of OAQ B do not intersect, the blocks must be tested to ensure whether they are disjoint, coincident or one block lies inside the other.

This action is done in procedure CHECK_BLOCKS. The test is based on the width of the blocks of OAQ A and OAQ B. The width of the OAQ A's block is (2^{levelA}) and the width of the OAQ B's block is (2^{levelB}). If these values are the same, then a check must be made to determine whether they are coincident or disjoint. If they are coincident then the resulting polygon is one of them.

If the width of the block of OAQ A is less than the width of the block of OAQ B and the center of OAQ A's block lies inside (to the left) of the edges of OAQ B then the block of OAQ A is inside the block of OAQ B, otherwise it lies outside of OAQ B. The same method is taken when the width of a block of OAQ B is less then the width of of the block of OAQ A; the point to be tested is the center of the block of OAQ B.

After examining each pair of nodes of OAQ A and OAQ B, then if both blocks intersect there is a polygon extraction resulting from calling procedure ARBOR_-BLOCKS_INT (see subsection 3.2.1). If the blocks are coincident or one block lies inside the other procedure EXTR_BLOCK extracts one of the blocks which lies inside

the other or one of the coincident blocks as a resulting polygon of intersection between two blocks. This polygon extraction is done by forming a linked list of edges with their corresponding edge names. Each extracted polygon is stored in the visited black node of OAQ A.

The second step of the intersection operation is to merge the stored polygons. This is done by traversing the OAQ A in post order. When a traversal visits a BLACK node, the stored polygons are merged, and the resulting polygons are saved in the same node. If a traversal visits a GRAY node then the polygons stored in the children nodes are merged, and the resulting polygons are stored in the visited node. Finally, after visiting all nodes, the traversal visits the root node and merges all polygons stored in the children nodes. The final result is one or more polygons which can be represented into one or more OAQs. For example, the result of the intersection between the two OAQs shown in Figure 3.8 is a set of edges defining a polygon with corner points JKLMNO and it is shown in Figure 3.27. To represent this object into another OAQ, this object needs to be transformed to object space coordinates to facilitate the computation of the orientation of the object. This is done by applying the inverse of the orthogonal transformation of equation (2.40). The second order moments are calculated using the method discussed in section 2.1. The object now can be represented into another OAQ using the steps discussed in chapter 2. The top level algorithm for performing Boolean OAQ operation is shown in Figure 3.28.

3.5 Union operation between two OAQs

The union operation is defined as [33]

$$A \bigcup B = \{x | x \epsilon A \ \ or \ \ x \epsilon B\} \tag{3.14}$$

In this definition, "or" means x belongs to either A or B or both. The Venn diagram of A union B is shown in Figure 3.29. As we see in the diagram, the interior of the

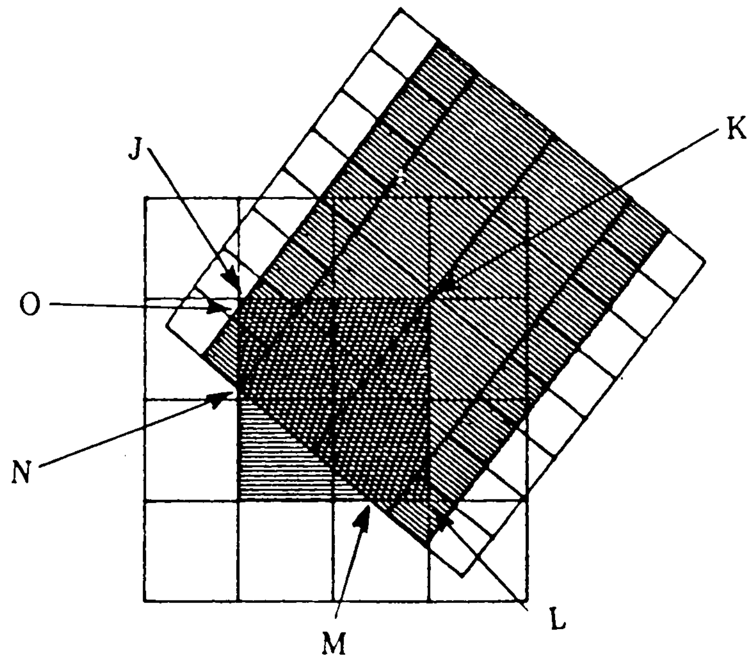


Figure 3.27: A set of edges defining the result of intersection between two OAQs.

```
ratio_A,ratio_B,t_AB: real; depth_A,depth_B : integer;
; history_A,history_B : [1..maxlevel] of byte

procedure BOOL_OAQ_OPER(var OAQ_A : OAQ; OAQ_B : OAQ);

var i : integer; Bool_Int,Bool_Union : boolean;
begin
    depth_A := 1; depth_B := 1;
    history_A[depth_A] := 0; history_B[depth_B] := 0;
    ratio_A := OAQ_A.sizi / OAQ_A.sizo; ratio_B := OAQ_B.sizi / OAQ_B.sizo;
    ratio_AB := ratio_A / ratio_B;
    maxlevel_A := MAX_DEPTH(2,OAQ_A.sizi);
    maxlevel_B := MAX_DEPTH(2,OAQ_B.sizi);
    if Bool_Int then begin
        INT_CHECK_NODES(OAQ_A.qt,OAQ_B.qt, maxlevel_A,OAQ_A.sizi,OAQ_A.sizi,
        maxlevel_B,OAQ_B.sizi,OAQ_B.sizi);
        MERGE_OPERATION(OAQ_A.qt); end;
    if Bool_Union then begin
        UNION_CHECK_NODES(OAQ_A.qt,OAQ_B.qt, maxlevel_A,OAQ_A.sizi,OAQ_A.sizi,
        maxlevel_B,OAQ_B.sizi,OAQ_B.sizi);
        MERGE_OPERATION(OAQ_A.qt); end;
    for k := 1 to OAQT1.qt^.count do begin
        ck := OAQ_A.qt^.poly_list[k].edge_list;
        { Transform object to object space coordinate }
        IMG_TO_OBJ(ck,corners,pts,theta_A,xc_A,yc_A);
        sizo := OBJECT_SIZE(corners,pts);
        readln(sizi);
        { Orientation calculation and OAQ representation }
        OAQ_REP(OAQ_A,pts,corners);
    end;
end; { of procedure BOOL_OAQ_OPER }
```

Figure 3.28: The BOOL_OAQ_OPER algorithm for doing Boolean OAQ operations.

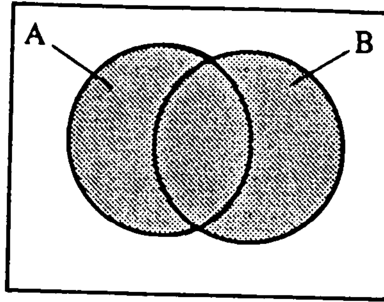


Figure 3.29: Union of A and B.

large rectangle represents the universe which is called U.

The complement of A in U is defined as

$$\overline{A} = U - A = \{x | x \epsilon U \ \ and \ \ x \notin A\} \tag{3.15}$$

The Venn diagram of $\overline{A}$ is depicted in Figure 3.30. Figure 3.31 shows the intersection

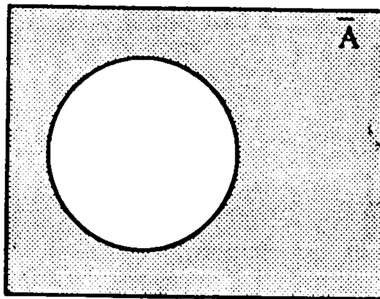


Figure 3.30: The complement of A.

of $\overline{A}$ and B. $A \bigcup B$ can now be defined as

$$A \bigcup B = A \bigcup (\overline{A} \bigcap B) \tag{3.16}$$

The Venn diagram of $A \bigcup (\overline{A} \bigcap B)$ is shown in Figure 3.32. The same concept is

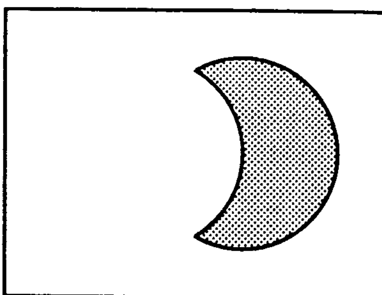


Figure 3.31: The intersection of the complement of A and B.

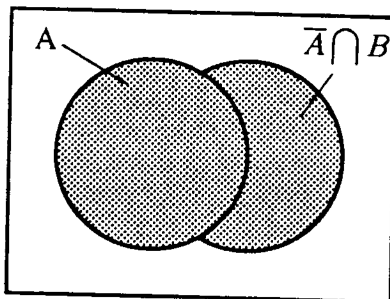


Figure 3.32: The union of A and $(\overline{A} \cap B)$.

adopted to compute the union of two OAQs. A and B represent the BLACK blocks of OAQ A and the BLACK blocks of OAQ B. $\overline{A}$ represents the WHITE blocks of OAQ A. Consider Figure 3.33. This figure depicts the subtree of a union operation

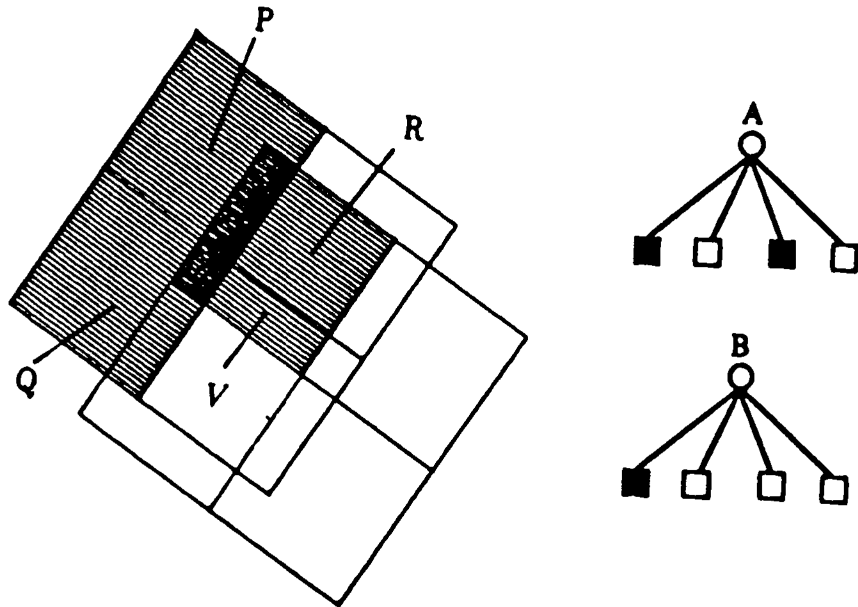


Figure 3.33: The subtree of union operation between two OAQs.

between two OAQs and the corresponding images. If a process of examining nodes visits a NW node of a subtree of OAQ A, the block representing the node is extracted and stored in the same node as a polygon called P. A similar action is taken when the process visits the SW node of a subtree of OAQ A. This polygon is denoted as Q in Figure 3.33. When the process of examining nodes meets the NE white node of subtree A, and the NW black node of subtree B, the intersection of the corresponding blocks is extracted and stored in the white node of subtree OAQ A, as a polygon called R. The same process is taken when the SE white node of subtree of OAQ A and the NW black of a subtree of OAQ B are met. The extracted polygon is denoted as V in the figure.

When the polygon extractions are complete, then OAQ A is traversed. When a traversal meets the root node of subtree OAQ A, then polygons P,Q,R and V which are stored in the NE, NW, SE, and SW nodes respectively, are merged and the result is a polygon defining the union operation between subtree of OAQ A and subtree of OAQ B.

Knowing the OAQ representation of two objects, and the above definition (3.16), the union of two OAQs can be computed by extracting the blocks of OAQ A and also extracting the intersection between white blocks of OAQ A and black blocks of OAQ B. All the intersections between white blocks of OAQ A and black blocks of OAQ B are equivalent to $\overline{A} \cap B$ in the equation (3.16).

The algorithm for computing the union operation between two OAQs is similar to the algorithm for computing the intersection, except that the process of examining nodes for the union operation do not extract the intersection between two black blocks representing black nodes of OAQs A and B. Instead, it extracts the black blocks of OAQ A and stores them in the nodes of OAQ A, and it also extracts the intersection of white blocks of OAQ A and black blocks of OAQ B and stores them in the nodes of OAQ A. This algorithm is shown in Figure 3.34. When a traversal of both OAQs in parallel is carried out, a test to determine wether two blocks representing the two visited nodes intersect or not is conducted in a similar way to the method mentioned for the intersection above. If the two blocks do not intersect then the same test for coincidence or inclusion is carried out. The process to extract the resulting polygon is identical to the manner discussed above for intersection.

Having the list of polygons stored in BLACK and WHITE nodes of OAQ A, the next step is traversing the reference OAQ A. When a traversal visits either a BLACK or WHITE node, the stored polygons are merged, and the resulting polygons are saved in the same node. If a traversal visits a GRAY node then the polygons stored in the children nodes are merged, and the resulting polygons are also stored in the same node. Finally, after visiting all nodes, the traversal visits a root node and merges all

```
procedure
    UNION_CHECK_NODES(q1 : pqtree; q2 : pqtree;level_A, x1,y1, level_B,x2,y2 : integer);


var i,k,num_edge : integer; polygon : edge_ptr;
block_intersect,found : boolean; disjoint,same_block,A_in_B,B_in_A : boolean;
begin
    found := disjoint := same_block := A_in_B := B_in_A := false;
    RECORD_CORNERS(level_A,x1,y1,level_B,x2,y2);
    if (q1^.colour = 'B') and (q1^.poly_count = 0) then begin
        same_block := true; A_in_B := false; B_in_A := false; new(polygon);
        { Extract block of OAQ A }
        EXTR_BLOCK(level_A,level_B,same_block,A_in_B, B_in_A,polygon); end;
    if (q1^.colour = 'W') and (q2^.colour = 'B') then begin
        if A_B_INTERSECT then begin { block of OAQ A and B intersect }
            new(polygon); ARBOR_BLOCK_INT(level_A,level_B,polygon,num_edge);
            found := false else found := true; end
        else begin { A = B or A inside B or B inside A }
            CHECK_BLOCKS(disjoint,same_block,A_in_B,B_in_A, level_A,level_B);
            if disjoint then found := false else begin
                EXTR_BLOCK(level_A,level_B,same_block, A_in_B,B_in_A,polygon);
                found := true; end;
    end; end;
    if found then begin { store a polygon in the node of OAQ A }
        with q1^ do begin
            poly_count := poly_count + 1; POLY_LIST[poly_count].edge_list := polygon;
            POLY_LIST[poly_count].depth := depth_B;
            POLY_LIST[poly_count].history := history_B; end; end;
    if (q1^.colour = 'W')and (q2^.colour = 'G') then
        for the four children of q1 do call UNION_CHECK_NODES;
    if (q1^.colour = 'G') and (q2^.colour = 'B') then
        for the four children of q1 do call UNION_CHECK_NODES;
    if (q1^.colour = 'G') and (q2^.colour = 'G') then begin
        level_A := level_A - 1; level_B := level_B - 1; block_intersect := false;
        if A_B_INTERSECT then { block of OAQ A and B intersect }
        else CHECK_BLOCK(disjoint,same_block,A_in_B,B_in_A, level_A,level_B);
        if block_intersect or same_block or A_in_B or B_in_A then begin
            depth_A := depth_A + 1; depth_B := depth_B + 1;
            for the 4 children of q1 do
                for the four children of q2 do call UNION_CHECK_NODES; end
    end
    else if (level_A = maxlevel_A) and (level_B = maxlevel_B) then
            writeln('Images are completely disjoint ');
end. { of procedure UNION_CHECK_NODES }
```

Figure 3.34: The UNION_CHECK_NODES algorithm for extracting polygon for union Boolean operation.

polygon which are stored in the children nodes. This results in one or more polygons which can be represented into one or more OAQs. The method of merging polygons has been discussed in section 3.2.

The union operation of two OAQs involves the extraction of black blocks of OAQ A and all pairs of intersections between white blocks of OAQ A and black blocks of OAQ B. The black blocks of OAQ B which lie outside the root block of OAQ A need also to be considered. Consider Figure 3.35. The part of the black blocks of OAQ B

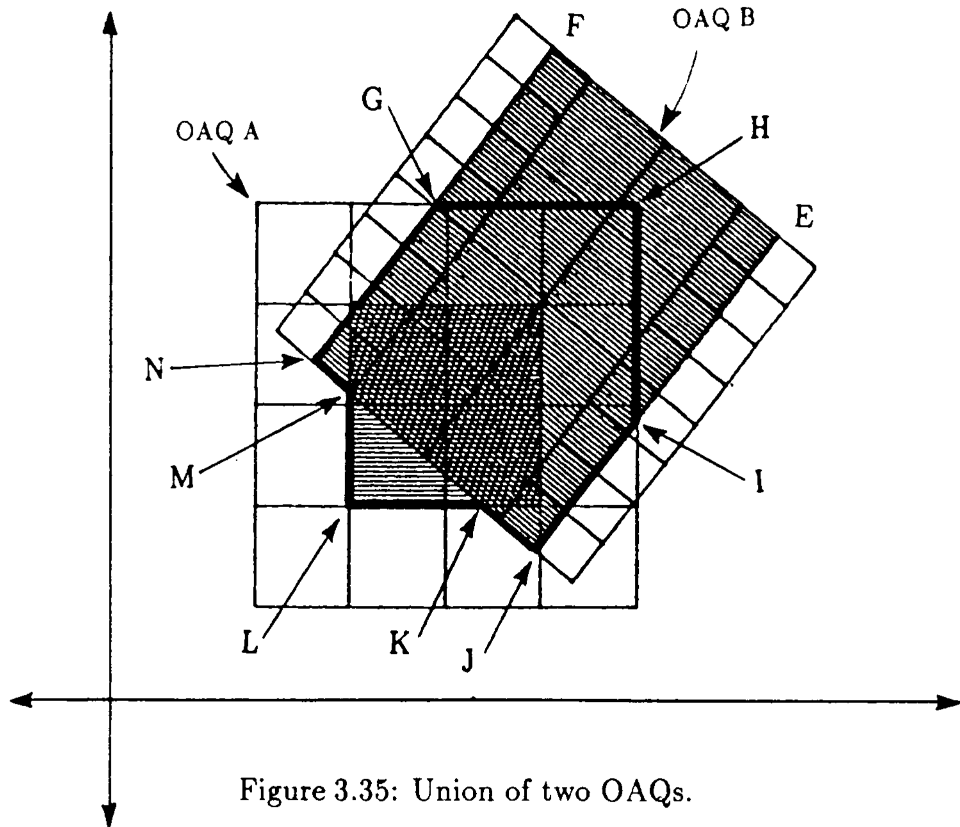


Figure 3.35: Union of two OAQs.

which lie outside the root block of OAQ A is a polygon whose vertices are denoted by I E F G H. This portion should be included in the union operation between the OAQs in order to get the correct result. Consider a union operation between the OAQs shown in Figure 3.35. The result of the union operation between these OAQs is a polygon bounded with the darker line. The union operation only results in a polygon which lies inside the root block of OAQ A (in this case, the polygon with

corner points G,H,I,J,K,L,M,N). This polygon is not the correct result. To get the correct result, the OAQ A needs to be expanded to cover all black blocks of OAQ B. This is done by enlarging the object space size of OAQ A.

Figure 3.36 depicts the result of the union operation when OAQ A is properly enlarged. The shaded polygon shown in the figure is the result of the union operation.

The analysis of the complexity of the algorithms for doing set theoretic operations of two OAQs will be discussed in the next chapter.

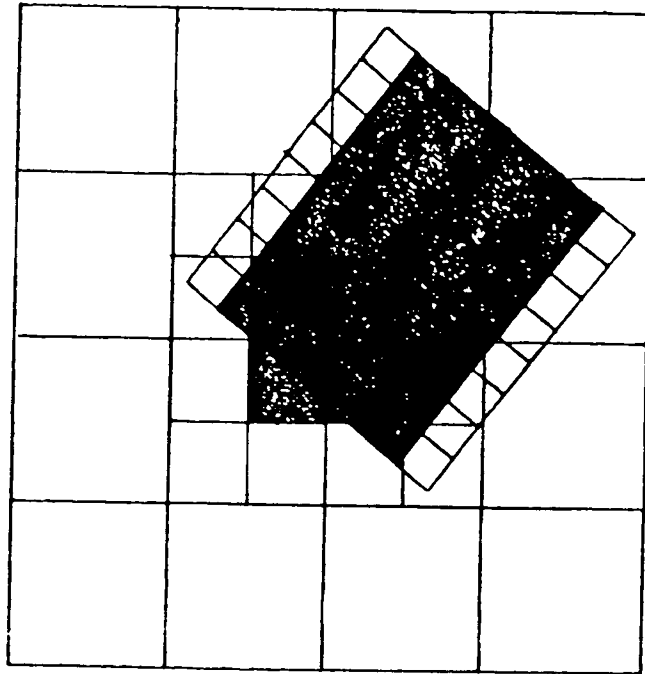


Figure 3.33: The result of Boolean union operation of two OAQs from Figure 3.32.

Chapter 4

Assessment

The work of this thesis has been assessed using data from a digital dataset representing a 1:50,000 scale map of the Fredericton area. The test was also augmented with simulated data in order to test the other cases. This assessment was based on the requirement for a suitable comparison of computing the number of nodes in both Standard and OAQ quadtrees. The selection requirement was based on placing the standard quadtree axes in the object's centroid, since one of the quadtree's features is that they are variant under translation and rotation, and the OAQ property is that the OAQ axes are aligned with the principle axes of inertia of the objects.

4.1 Simulated test cases

The long skinny object shown in Figure 4.1 was encoded using both OAQs and standard quadtrees. This object was rotated by increments of -10° from 0° to $-50°$. Image space size of 16 by 16 up to 512 by 512 were used to encode the object. The result are tabularized and shown in Table 4.1. Note that the different number of nodes resulting for the OAQ representations at an image size of 16 are due to rounding errors when computing the integer coordinates required by the Bresenham algorithm. Other simulated data have been encoded into OAQ representations. The results of

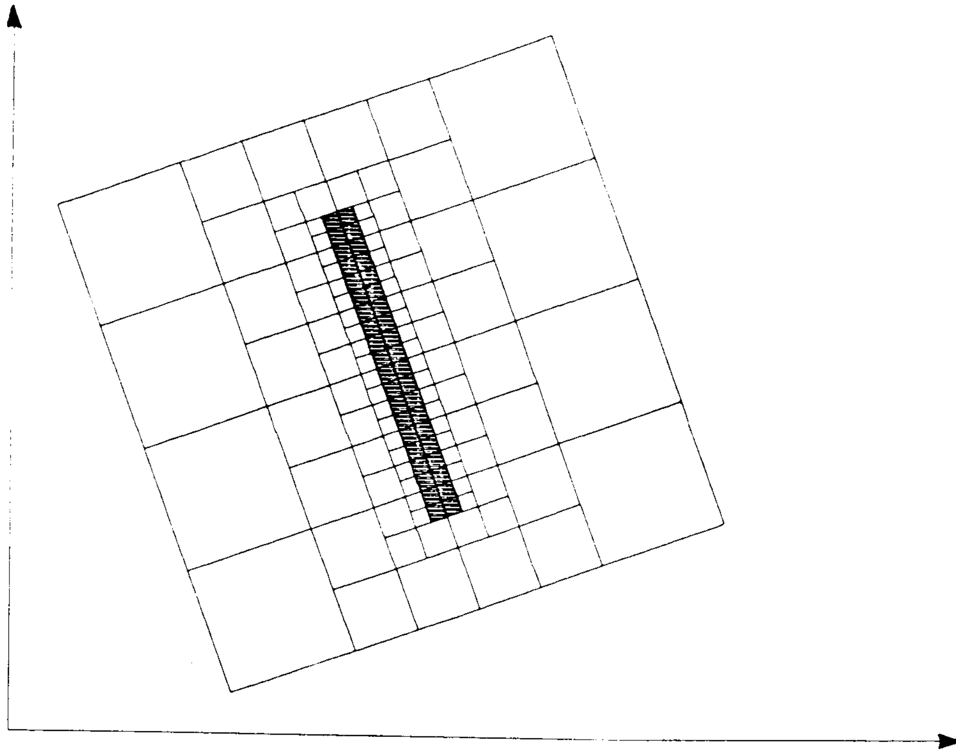


Figure 4.1: Example of an object encoded as OAQ, original object corner coordinates are (-0.5,5), (0.5,5), (0.5,-5), (-0.5,-5).

OAQ	Image Size					
θ	16x16	32x32	64x64	128 x 128	256 x 256	512x512
0	65	181	181	181	181	181
-10	65	181	181	181	181	181
-20	77	181	181	181	181	181
-30	77	181	181	181	181	181
-40	45	181	181	181	181	181
-50	65	181	181	181	181	181
St.Q	**Image Size**					
θ	16x16	32x32	64x64	128 x 128	256 x 256	512x512
0	53	181	181	181	181	181
-10	73	141	245	525	1053	2117
-20	65	131	285	573	1149	2341
-30	73	141	309	613	1237	2493
-40	61	101	333	621	1261	2597
-50	69	137	317	613	1285	2565

Table 4.1: Number of nodes for Figure 4.1 object encoded as an OAQ and as a standard quadtree (St.Q).

OAQ representations of the two objects with different object to image scale factors are shown in Figures 4.2 and 4.3. These OAQ representations have been tested to compute the Boolean OAQ operations. The result of computing the intersection has been encoded into an OAQ and is shown in Figure 4.4, and the result of the union operation is shown in Figure 4.5. The result of Boolean OAQ operations can be encoded into another OAQ which may differ in resolution.

Table 4.2 shows the time response of computing the Boolean OAQ operation of OAQs of Figure 3.8 with object space sizes of OAQ A and B is 16. The image space size varies from 16 up to 256.

Image size	Intersection	Union
	second	second
16 x 16	1.43	3.19
32 x 32	1.85	5.10
64 x 64	2.25	7.25
128 x 128	2.70	9.12
256 x 256	2.87	10.54

Table 4.2: The execution time of Boolean OAQ operations of Figure 3.8.

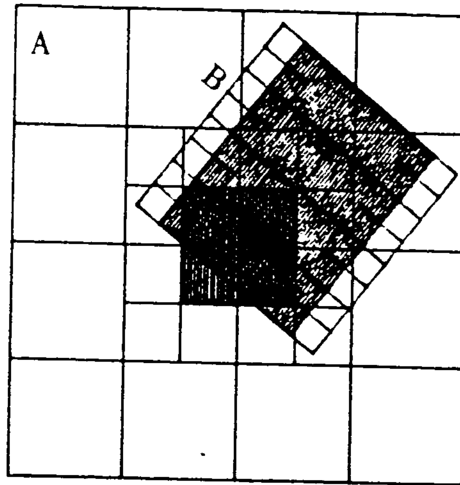
4.2 Real test cases

Several data have been extracted from Fredericton map data, and they have been encoded using OAQ and standard quadtrees. The result are tabularized and shown in Table 4.3. For the linear road objects, areas were formed by tracing a polygonal band around the center-line [17].

In summary, the average savings in the number of nodes required for the OAQ over all 17 objects in Table 4.2 are 8.5 % for 128 $\times$ 128 image size, and 9.5% for 256 $\times$ 256 image size compared to the standard quadtree.

The Boolean OAQ operations approach have been assessed using both simulated data discussed above, and data which was extracted from the Fredericton map dataset.

Regions A and B

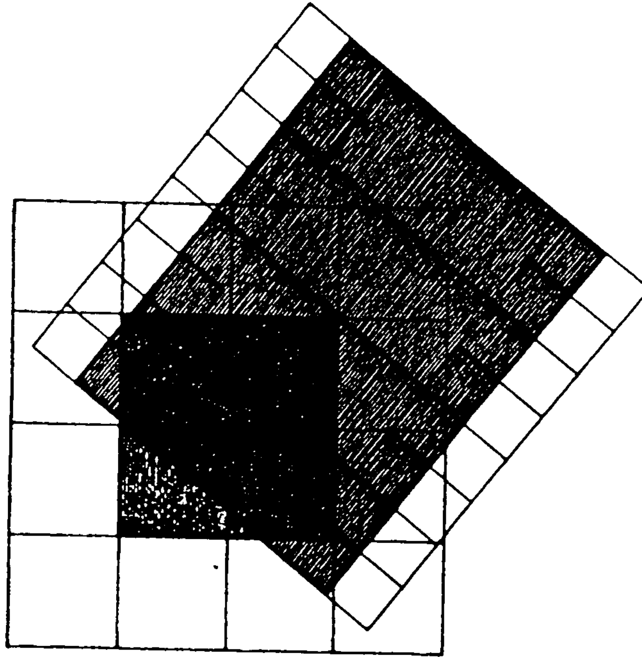


OAQ A	
Nodes	: 37
White	: 24
Black	: 4
Theta	: 0.00
O Size:	32
I Size:	16
Time	: 0: 0: 0.11

OAQ B	
Nodes	: 53
White	: 16
Black	: 24
Theta	: -38.50
O Size:	16
I Size:	16
Time	: 0: 0: 0.11

Figure 4.2: OAQ representation of the objects A and B with object to image scale factors of A = 2 and B = 1.

Regions A and B



OAQ A		OAQ B	
Nodes	: 21	Nodes	: 53
White	: 12	White	: 16
Black	: 4	Black	: 24
Theta	: 0.00	Theta	: -38.50
O Size:	16	O Size:	16
I Size:	16	I Size:	32
Time	: 0: 0: 0.11	Time	: 0: 0: 0.22

Figure 4.3: OAQ representation of the objects A and B with object to image scale factors of A = 1 and B = 0.5.

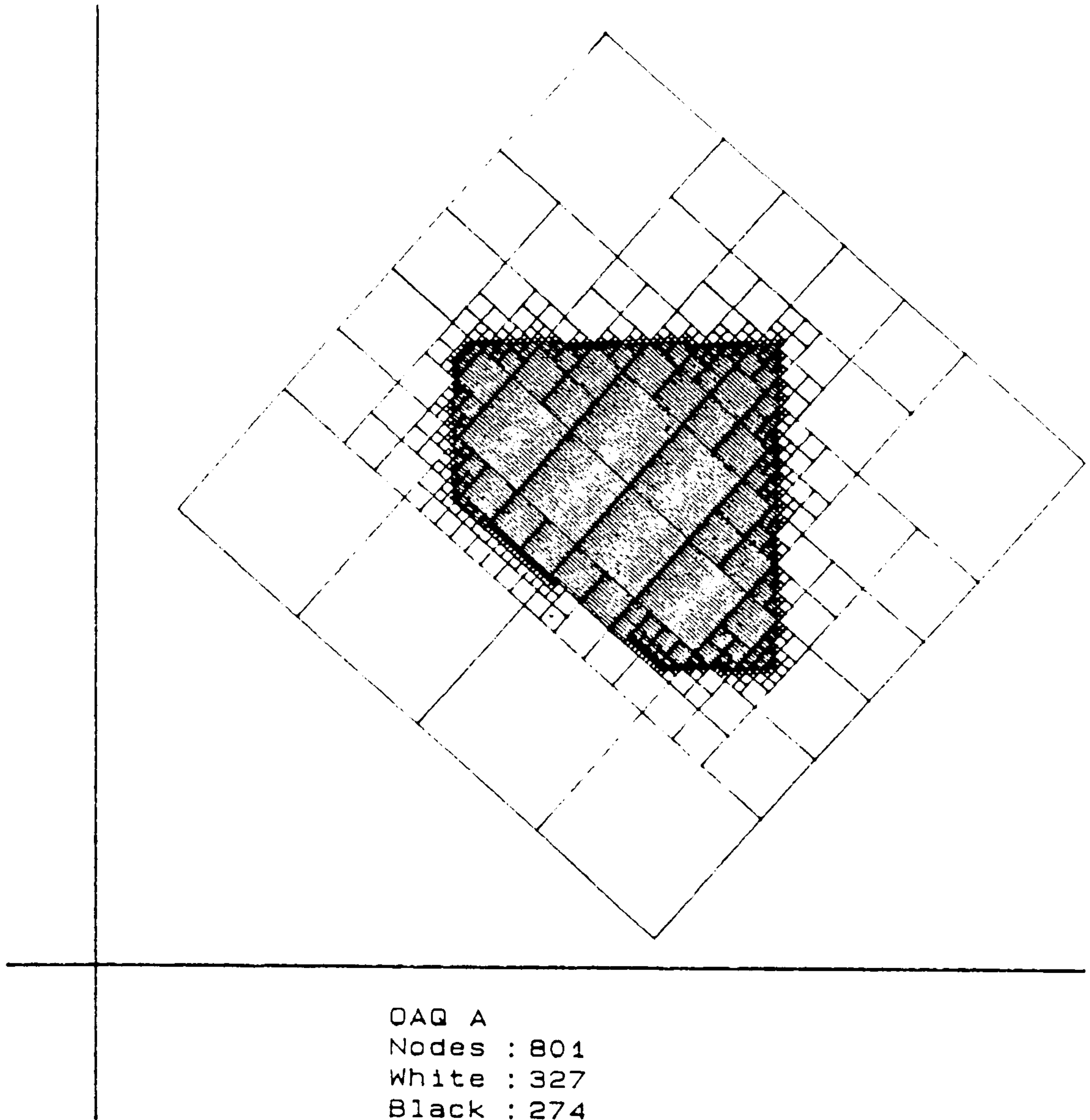


Figure 4.4: The result of the Boolean intersection operation for OAQs A and B of Fig.4.3.

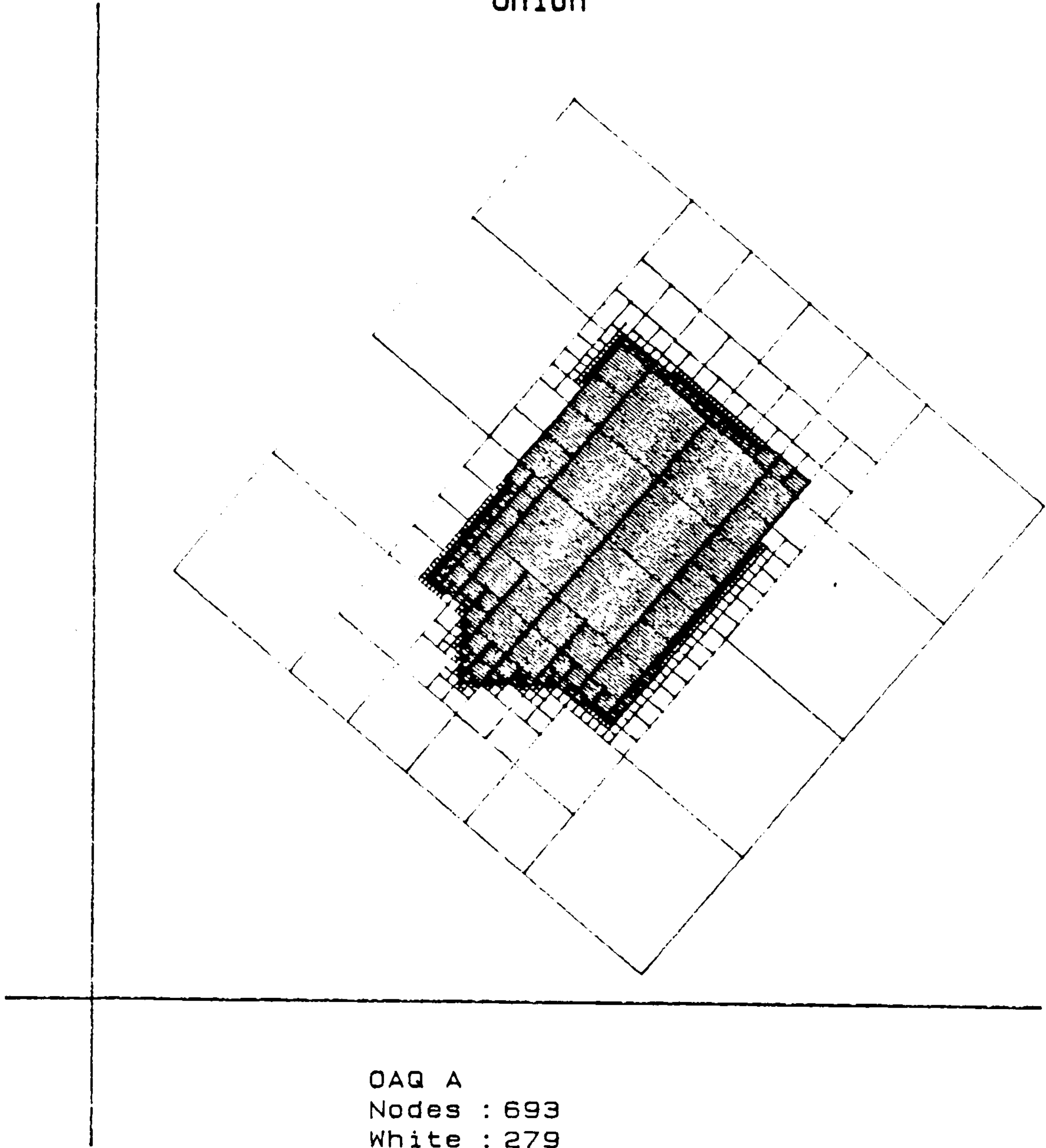


Figure 4.5: The result of the Boolean union operation for OAQs A and B of Fig.4.3.

Name (location)	*OAQ (θ)*	*Size image*	*OAQ total nodes*	*STQ total nodes*	*OAQ leaf nodes*	*STQ leaf nodes*	*OAQ time second*	*STQ time second*
Ball Park (French Village)	35.09	128 x 128	509	621	382	446	2.53	3.19
		256 x 256	1069	1297	802	973	16.70	21.37
Park (Silverwood)	-5.95	128 x 128	665	667	492	508	2.14	2.36
		256 x 256	1389	1365	1042	1024	12.58	13.90
Park (Skyline)	5.36	128 x 128	733	745	550	559	2.59	2.75
		256 x 256	1557	1517	1168	1138	15.88	17.69
Park (Devon)	17.08	128 x 128	657	761	493	571	2.96	3.24
		256 x 256	1349	1485	1012	1114	18.96	21.37
Park (South Devon)	-69.65	128 x 128	785	969	589	727	5.22	5.16
		256 x 256	1565	1957	1174	1464	36.73	36.30
Officer Square (Fredericton)	-78.52	128 x 128	709	725	532	544	3.62	3.84
		256 x 256	1417	1565	1063	1174	24.11	26.80
Park (Marysville)	-83.7	128 x 128	1097	1029	823	772	5.44	5.33
		256 x 256	2229	2209	1672	1657	37.62	37.64
Park (Lincoln St)	-78.90	128 x 128	913	1009	685	757	4.83	4.99
		256 x 256	1853	1937	1390	1453	34.58	34.82
Park (Oromocto)	64.62	128 x 128	501	641	376	481	3.37	3.52
		256 x 256	1113	1337	835	1003	19.94	23.67
Ball Park (Oromocto)	41.30	128 x 128	837	881	628	661	2.91	3.37
		256 x 256	1589	1853	1192	1375	3.52	4.45
Queen Square (Fredericton)	-41.30	128 x 128	593	665	445	514	2.91	3.37
		256 x 256	1189	1365	892	1024	18.66	22.69
Airfield (Fredericton)	1.16	128 x 128	453	477	340	358	1.13	1.16
		256 x 256	629	789	472	592	4.57	4.97
Kelly Creek Basin	7.64	128 x 128	457	505	373	379	3.69	3.62
		256 x 256	1077	1049	808	787	14.63	15.53
Tower Lake	-60.39	128 x 128	909	1093	682	820	6.34	5.78
		256 x 256	1929	2189	1447	1642	43.47	38.72
Scotch Lake	-11.07	128 x 128	1163	1161	871	871	6.87	6.54
		256 x 256	2481	2485	1861	1864	44.38	42.24
Road (1)	-57.33	128 x 128	725	793	544	595	4.95	4.56
		256 x 256	1441	1625	1081	1219	34.39	30.96
Road (2)	-28.53	128 x 128	589	633	442	475	4.39	4.12
		256 x 256	1061	1301	796	976	30.15	27.41

Table 4.3: Encoding objects using OAQs and standard quadtrees.

To carry out Boolean OAQ operations with real data, two sets of data are overlapped, and encoded into OAQs.

Two different objects (one representing a forest area, the other being Tower lake) were taken from the Fredericton map data set and overlapped by shifting the data so that one intersects the other. Figure 4.6 shows the objects to be represented as OAQs, and Figure 4.7 depicts the OAQ block decompositions of the two objects, where one has been overlapped with the other. After having the data sets, the intersection and union operations are computed and the results are shown in Figures 4.8 and 4.9.

The work of this thesis was coded using the TURBO PASCAL 5.0 on an IBM Personal System/2 Model 70 386, and the code is about 10,000 lines. Since the maximum heap can only be set to 64KB, the encoding of real data into both OAQs and quadtrees can only reach a maximum size image of 256×256; otherwise the program will not execute properly.

4.3 Analysis

Quadtrees provide a relatively compact representation for regions. They also provide set theoretic operations, such as union, and intersection. Boolean quadtree algorithms are simply preorder traversals of normal quadtrees, and the execution time is generally a linear function of the number of nodes in the quadtree. It does not depend on the size of the image. This section discusses the execution time analysis of OAQ algorithms.

The most natural way to analyse the execution time of the functions involved both in OAQ constructions and Boolean OAQ operations is in terms of the number of nodes that must be visited during the operations.

The algorithm for converting a planar object into an OAQ depends on the number of times procedure CONSTRUCT is invoked (see Figure 2.22). Each pixel is recursively inspected by inference from the RLC description. The execution time of the algorithm is proportional to the number of pixels in the image, $O(n^2)$, where $n =$

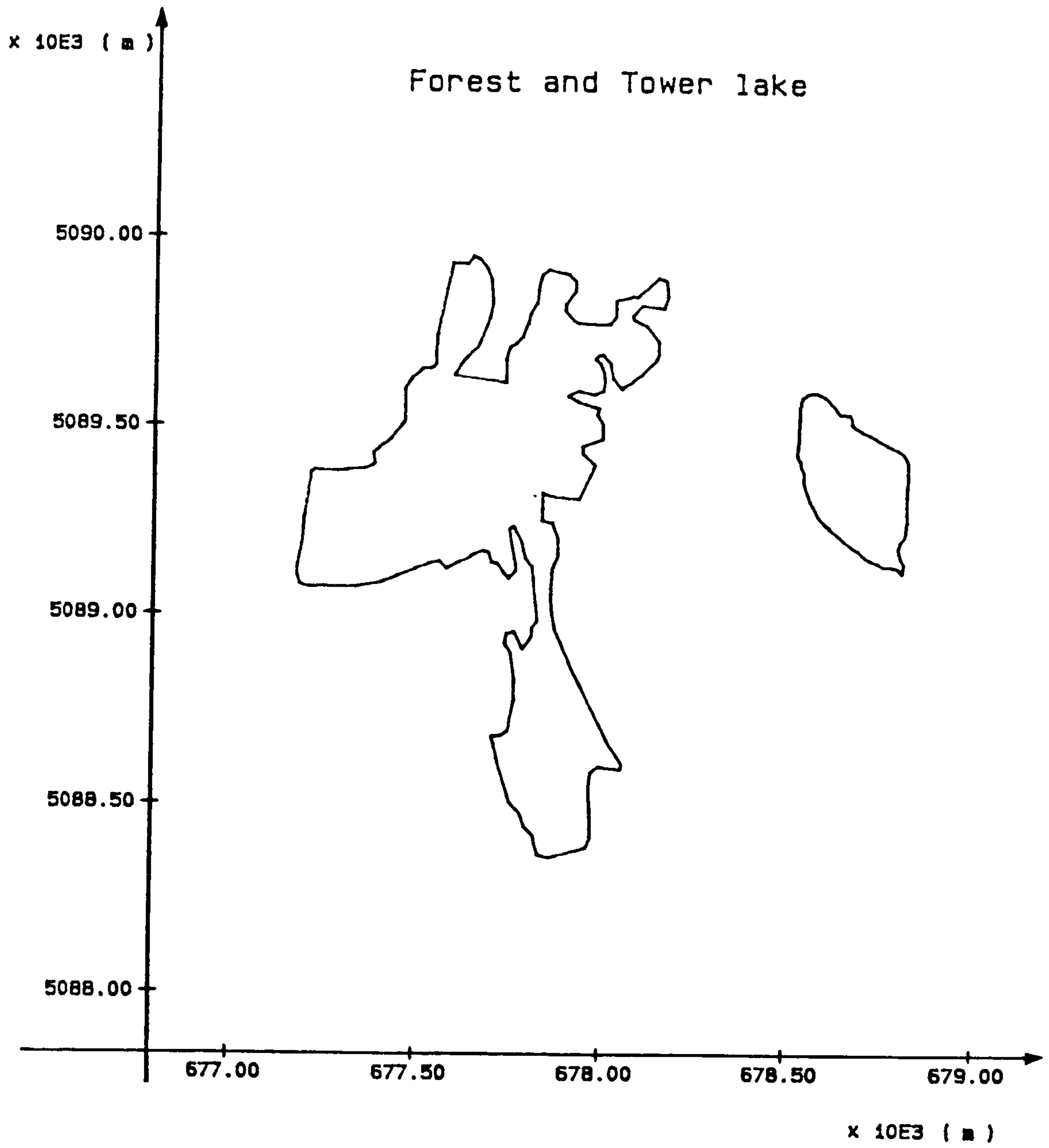
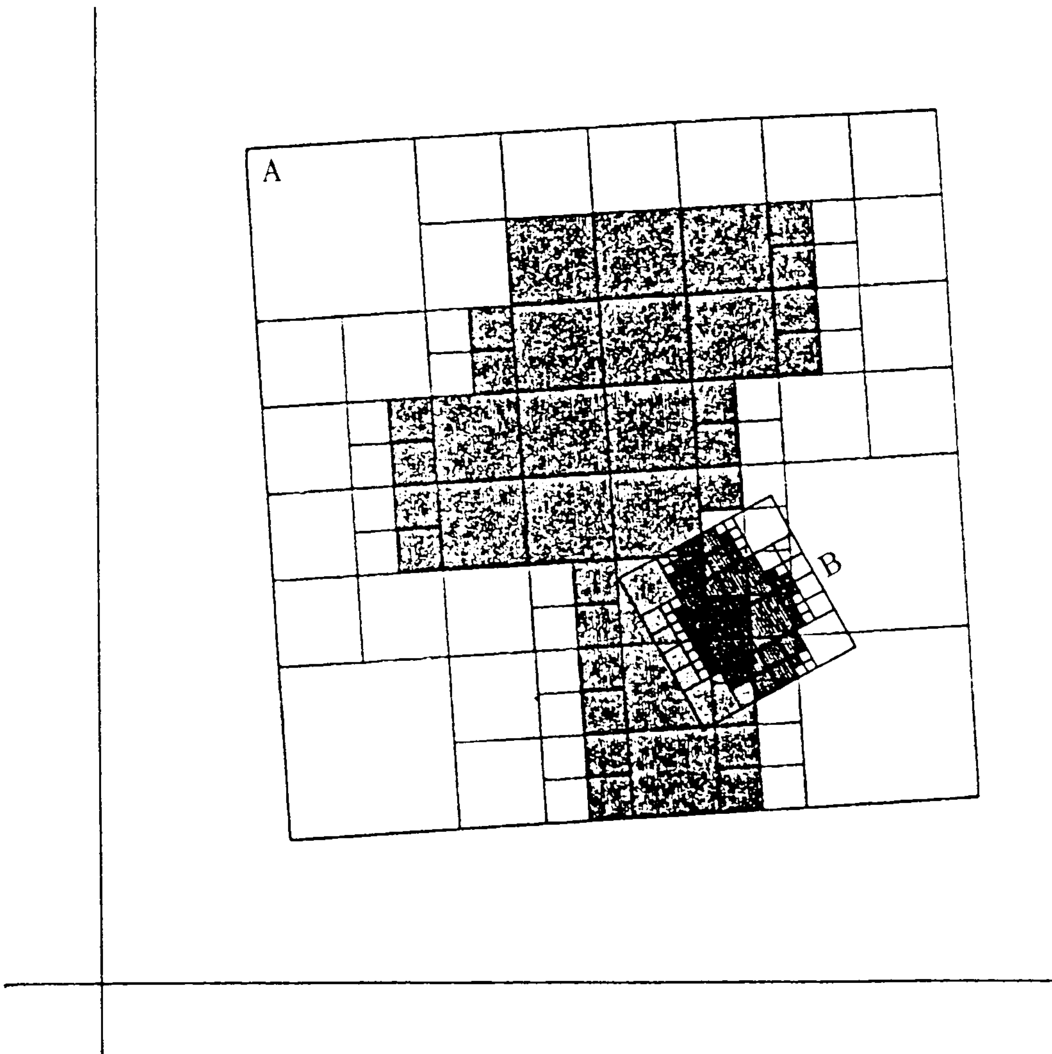


Figure 4.6 The original objects of a forest area and Tower lake.



OAQ A
Nodes : 121
White : 52
Black : 39
Theta : -86.58
O Size: 2048
I Size: 16
Time : 0: 0: 1.48

OAQ B
Nodes : 97
White : 38
Black : 35
Theta : -61.01
O Size: 512
I Size: 16
Time : 0: 0: 0.55

Figure 4.7: The OAQ representations of the forest and the lake.

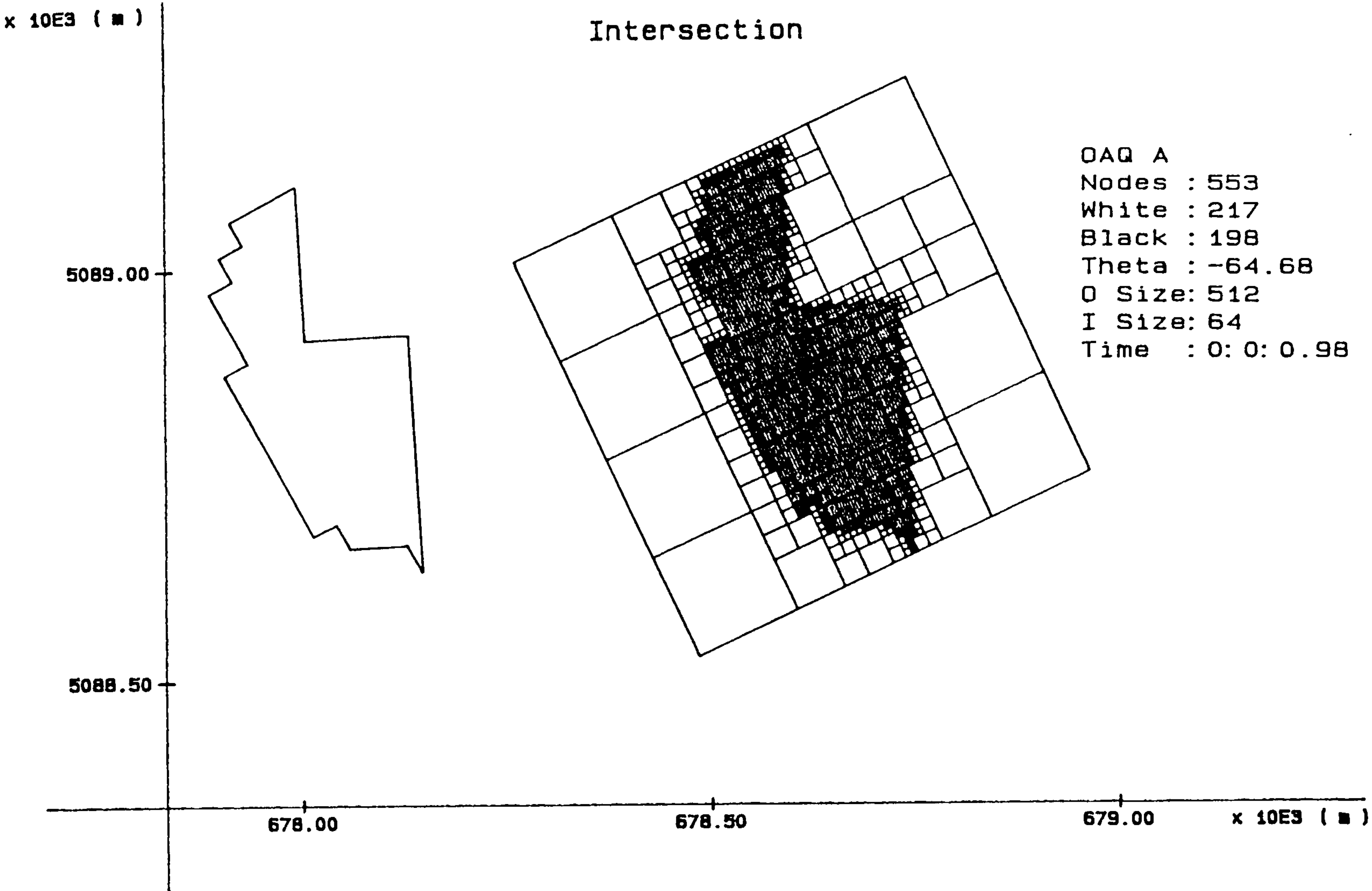


Figure 4.8: The result of the intersection OAQ operation of of Figure 4.7, and its OAQ representation.

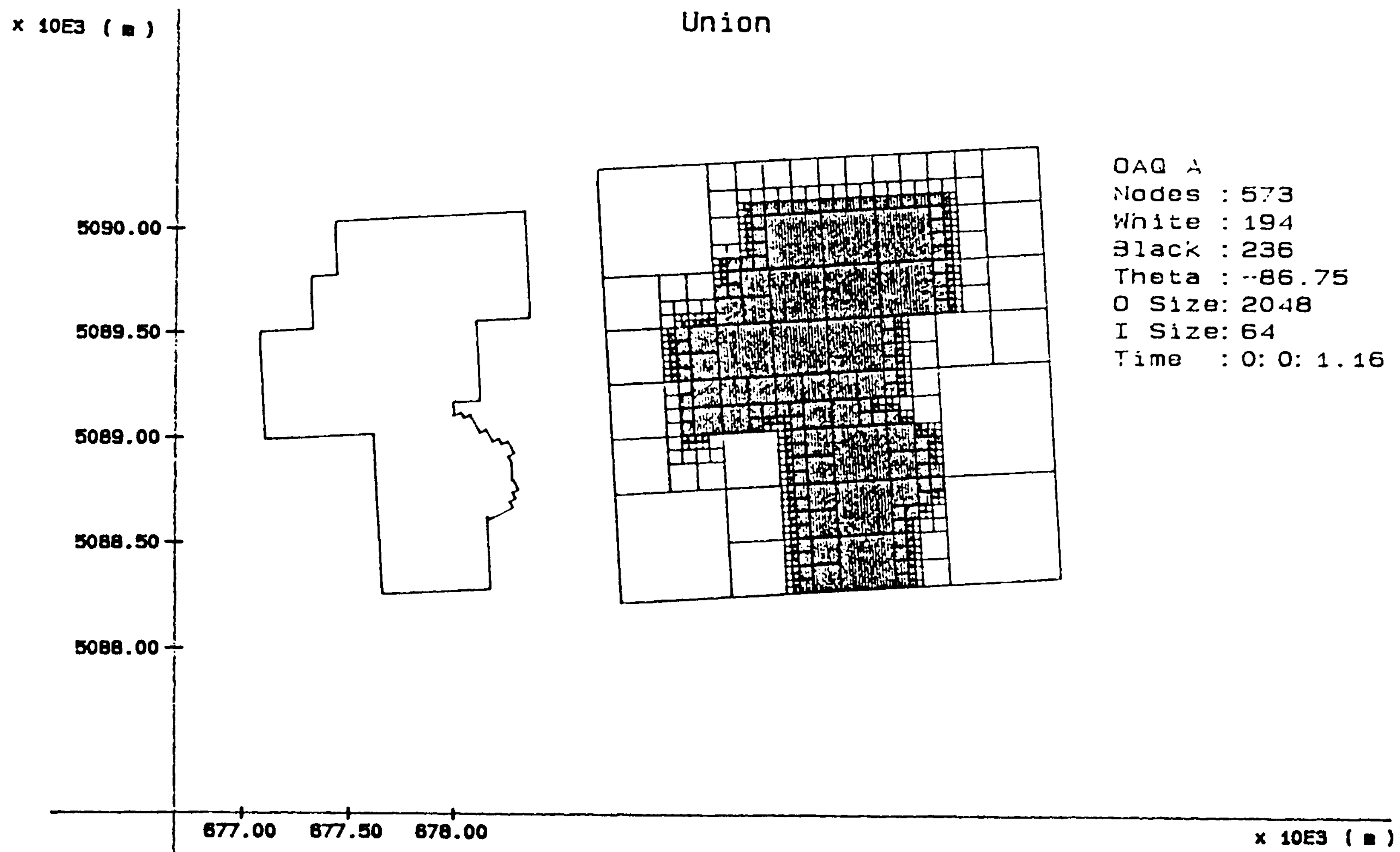


Figure 4.9: The result of the union OAQ operation of Figure 4.7, and its OAQ representation.

image space coordinate range [26]. The computation of the orientation of the object is proportional to the number of edges in the original object description [18]. This extra time does not affect the time efficiency of OAQ construction.

OAQ representation makes use of second order moments to reduce the number of nodes. In the standard quadtree, the number of nodes representing a region is linearly proportional to the resolution of the image. Table 4.3 shows that doubling the resolutions, leads to a significant increase in the number of nodes. In most OAQ representations, if the resolution doubles and hence the number of nodes is less than that of the standard quadtree, it may be constant for varying resolutions and orientations (see Table 4.1).

Quadtree representations allow set theoretic operations to be performed efficiently. For standard quadtrees, performing set operations of two aligned quadtrees is simple to implement. Both of the quadtrees are traversed in parallel, each step in the traversal investigating the colour of both nodes in order to decide the colour of the corresponding node in the resulting quadtree. The worst-case execution time of the algorithm for doing set theoretic operations, reviewed in chapter 3, is proportional to the sum of the number of nodes in the two input quadtrees [25]. This algorithm does not provide a way to perform boolean set theoretic operations of quadtrees representing objects which are not aligned.

OAQ representation provides a way to perform set theoretic operations between two differently oriented regions with different OAQ centers, and different object to image scale factors. As mentioned in chapter 3, the Boolean OAQ operation requires steps of extracting polygons produced by intersection of pairs of BLACK nodes of different OAQs, or pairs of WHITE nodes of OAQ A and BLACK nodes of OAQ B, and a step of merging those polygons which can result in polygons to be represented as other OAQs. The top level algorithm for doing Boolean OAQ operations is shown in Figure 3.28.

The step of extracting polygons requires a traversal of both OAQs in parallel

which is done in procedure INT_CHECK_NODES or UNION_CHECK_NODES. The examination of all pairs of nodes may be required to extract the polygons resulting from Boolean operations between two nodes from different OAQs. The most natural way to analyse the execution time of Boolean operations is in terms of the number of nodes that must be visited while computing the desired set operation between two blocks representing two specified nodes from different OAQs.

Procedures INT_CHECK_NODES, and UNION_CHECK_NODES shown in Figures 3.25 and 3.34 of section 3.4 and 3.5 represent the implementation of steps of extracting polygons of Figure 3.11. For the intersection or union operation, if both nodes of OAQ A and B are GRAY and they intersect, the algorithm is applied recursively to the children of the nodes of A and B to test whether two nodes of OAQ A and B intersect or not. Assuming that the children of nodes of A and B have GRAY nodes, and each pair of GRAY nodes being tested intersect, the algorithm is applied again to the children of OAQs A and B. The process of examining nodes to know whether the corresponding blocks intersect or not requires a traversal two OAQs in parallel and block intersection test for pairs of GRAY nodes and pairs of BLACK nodes. This process will reach the worst-case whenever all pairs of the GRAY nodes of OAQ A and B intersect.

The algorithm for examining nodes to extract the intersection between two nodes takes advantage of the fact that if two GRAY nodes from different OAQs do not intersect then the algorithm is not applied to the subtree of those nodes. In this case, the number of operations will decrease significantly. In most cases of boolean OAQ operations, the time efficiency will not reach the worst-case execution, since not all nodes in the same level are GRAY nodes, and also not all pairs of GRAY nodes intersect.

The best case happens when two OAQ roots do not intersect, then no further action is taken. The execution time is proportionally to the constant value 1, $\Omega(1)$.

4.3.1 Analysis of intersection operation

As mentioned above, the process of examining nodes to know whether blocks intersect or not requires a traversal two OAQs in parallel and block intersection tests for pairs of GRAY nodes and pairs of BLACK nodes.

Consider the process of examining nodes for performing Boolean intersection operation shown in Figure 3.25. Since the block intersection tests are carried out only for pairs of BLACK blocks and pairs of GRAY blocks, and no action is taken when a WHITE block of OAQ A is visited, then the process requires $O((G_A + B_A) \times N_B + W_A)$ time where G_A, B_A, and W_A are the the number of GRAY, BLACK, and WHITE nodes of OAQ A respectively, and N_B is the number of nodes of OAQ B.

If the nodes being examined are BLACK nodes then a test is carried out to know whether they intersect or not. This is done by calling function A_B_INTERSECT, since the test is based on the line intersection test, where each edge of the block of OAQ B is tested against four successive edges of OAQ A until one intersection is found. The worst-case execution time of the algorithm is a constant value, denoted as j_1 (i.e. 16 line segment intersection tests).

After knowing that the block representing the BLACK nodes intersect, then the polygon extraction is carried out by calling procedure ARBOR_BLOCK_INT shown in Figure 3.7. The worst-case complexity of the algorithm is a constant value (8 output edges), and denoted as j_2.

To extract the intersection of two BLACK blocks representing two nodes, function and procedure A_B_INTERSECT and ARBOR_BLOCK_INT are called. Their execution time is a constant value $j = j_1 + j_2$. Supposing k_1 is the number of intersections of pairs of BLACK blocks, then extracting all the corresponding polygons requires $O(k_1 \times j)$ time. This execution time is proportional to the number of pairs of BLACK nodes intersecting, $O(k_1)$. In the worst case, all BLACK blocks of A intersect BLACK blocks of B.

If the two BLACK nodes being tested do not intersect, the corresponding blocks

are checked to know whether they are concident or not, or one block lies inside the other. This is done by calling procedure CHECK_BLOCKS shown in Figure 3.25. The test is based on the level number of OAQs and the block centers. One of the centers is tested to see whether it lies to the left of all four edges of the other block. The execution time of the algorithm is a constant value denoted as l_1.

When a test is complete, then the block which lies inside or coincides with the other is extracted by calling procedure EXTRACT_BLOCK whose execution time is a constant value denoted as l_2. After knowing the location of two BLACK nodes by traversing both OAQs in parallel, to extract the intersection of two BLACK nodes from two blocks which are coincident or one lies inside the other, the procedure CHECK_BLOCKS and EXTRACT_BLOCK are called. The execution time is a constant value $l = l_1 + l_2$. Suppose k_2 is the number of pairs of BLACK blocks which are coincident or where one block lies inside the other. Extracting all the polygons resulting from the blocks which are coincident or where one lies inside the other requires $O(k_2 \times l)$ time. This execution time is proportional to the number of pairs of BLACK nodes which are coincident or where one block lies inside the other. Since the value of k_2 is a constant, the execution time is denoted as $O(k_2)$.

Once the polygon extractions are complete, in which the algorithm requires $O((G_A + B_A) \times N_B + W_A + k_1 + k_2)$ time, then the OAQ reference is traversed in post order to merge the stored polygons.

4.3.1.1 Method I

The merging operations of method I discussed in section 3.3.1 has the algorithm shown in Figure 3.12. This algorithm shows that when a traversal visits leaf nodes, (only BLACK nodes for intersection), then the stored polygons are merged by calling procedure MERGE_OPERATION_B. When a traversal visits GRAY nodes then the algorithm merges the polygons which are stored in the GRAY children nodes. Thus the execution time of the merging polygons algorithm in terms of the number of nodes

that must be visited during operation is proportional to the number of nodes in OAQ A.

The proportional constant is determined by the time complexity of the procedures MERGE_OPERATION_B and MERGE_OPERATION_A. The first procedure is shown in Figure 3.13. The time required to merge all polygons stored in any BLACK node of OAQ A is $O(s + p)$, where s is the time required for sorting the list of polygons which are stored in the BLACK leaf node, and p is the time required for merging polygons which are stored in the BLACK leaf node which is proportional to the number of polygons. MERGE_OPERATION_B requires sorting processes, the first sorting is used to group polygons which belong to the same level of nodes of OAQ B. The second one is used to sort the polygons which belong to the same parent nodes of OAQ B. The MERGE_OPERATION_A algorithm shown in Figure 3.14 requires $O(r)$ time, where r is the number of polygons which are stored in the GRAY children nodes.

Denote N_A as the number of nodes of OAQ A, then the MERGE_OPERATION algorithm requires $O(N_A + B_A(p + s) + G_A r)$ time.

The worst-case execution time of the algorithm to carry out Boolean OAQ intersection operation using method I is $O(e + m_1)$, where

$$e = (G_A + B_A) \times N_B + W_A + k_1 + k_2 \tag{4.1}$$

and

$$m_1 = N_A + B_A(s + p) + G_A r \tag{4.2}$$

is the worst-case execution time of merging polygons using method I.

4.3.1.2 Method II

The algorithm for Boolean OAQ operation using method II is similar to method I, except for the process of merging polygons. The process of extracting polygons is the same as the one which is used in the method I. The process of merging polygons

is similar to the one shown in Figure 3.12 except for calling procedures CREATE_SUBTREE and MERGE_SUBTREE shown in Figures 3.18 and 3.19 rather than calling procedure MERGE_OPERATION_B shown in Figure 3.14. Therefore the algorithm of merging polygons does not require any sorting; instead, it makes a temporary subtree of OAQ B which have nodes intersecting with the visited BLACK node of OAQ A (see Figure 3.17). When OAQ A is traversed in post order, and a BLACK node is visited, then the algorithm creates the subtree of OAQ B, and it traverses the subtree in post order. If the traversal visits GRAY node of the created subtree, then all polygons which are stored in the children nodes are merged. The merging process is done in procedure MERGE_SUBTREE. This algorithm requires $O(t)$ times, where t is the number of polygons which are stored in the GRAY children nodes of subtree of OAQ B. The algorithm for creating the subtree requires $O(p)$ times, where p is the number of polygons which are stored in the BLACK node of OAQ A. The MERGE_OPERATION algorithm using method II requires $O(N_A + B_A(p+t) + G_A r)$ time.

The worst-case execution time of the algorithm to carry out Boolean OAQ intersection operation using method II is $O(e + m_2)$, where

$$e = (G_A + B_A) \times N_B + W_A + k_1 + k_2 \tag{4.3}$$

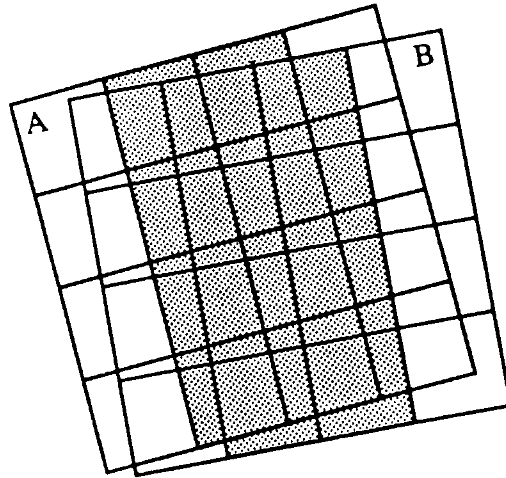
and

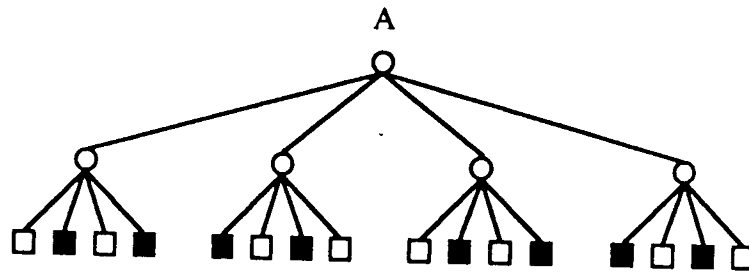
$$m_2 = N_A + B_A(p+t) + G_A r \tag{4.4}$$

is the worst-case execution time of merging polygons using method II. Overall, the worst-case execution time for performing Boolean OAQ intersection using method I or II is dominated by the time required for extracting polygons.

4.3.1.3 An example where all BLACK blocks intersect

This subsection gives an example of the analysis for the intersection operation. Consider Figure 4.10. This figure depicts the intersection between two OAQs. As





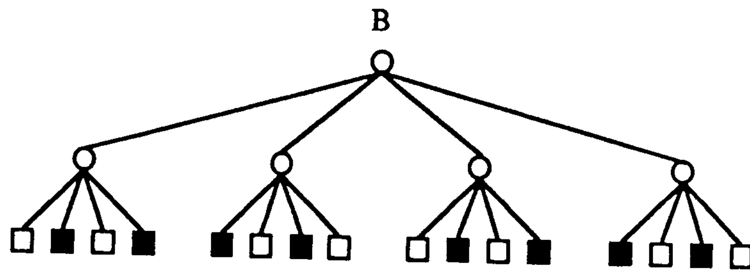


Figure 4.10: An example of the intersection operation.

mentioned above, the worst-case execution time for polygon extraction algorithm is $e = (G_A + B_A) \times N_B + W_A + k_1 + k_2$. Since all pairs of GRAY blocks of different OAQs intersect, and all pairs of BLACK blocks intersect, then G_A is total number of GRAY nodes of OAQ A $= 5$, and B_A is the total number of BLACK nodes of OAQ A $= 8$, $N_B = 21$, and $W_A = 8$. The term k_1 is the time required for extracting the intersection between two BLACK blocks which is proportional to the number of pairs of BLACK nodes intersecting. This example shows that each BLACK block of OAQ A intersect with at most 4 BLACK blocks of OAQ B. Thus we assume that in the worst-case each BLACK node of OAQ A stores 4 polygons. Since there is no BLACK block of OAQ A that lies inside the other or vise versa, then $k_2 = 0$. The algorithm for extracting polygons requires $(5 + 8) \times 21 + 8 + 4 = 285$ high-level operations.

The algorithm for merging polygons using method I requires $O(N_A + B_A(s + p) + G_A r)$ time. The term s is the time required for sorting the polygons which are stored at any BLACK node of OAQ A. This execution time is based on which sorting algorithm is used. Since the number of polygons is not large (4 polygons), a simple sort algorithm like the Bubble Sort can be chosen.This algorithm requires $O(q^2)$ time, where q is the number of elements being sorted. In this case, the number of polygons to be sorted are $p = 4$, thus $s = 4^2$. Once polygons in the visited BLACK node of OAQ A are sorted, these polygons are then merged, for which the algorithm requires $O(p)$ time. The term r is the time required to merging the merged polygons which are stored in GRAY children nodes, which is proportional to the number of merged polygons stored in the GRAY children nodes. In this example each GRAY child node has one merged polygon, $r = 1$. The worst-case algorithm for merging polygons using method I requires $21 + 8(4^2 + 4) = 181$ operation high-level. This worst-case execution time is dependent on the chosen sort algorithm, if the quick sort algorithm is used, the algorithm requires only $21 + 8(4log4 + 4) = 72$ high-level operations.

The algorithm for merging polygons using method II requires $O(N_A + B_A(p + t) + G_A r)$ time, where p is the time required for copying the subtree of OAQ B which its

nodes intersect with the visited node of OAQ A. This execution time is proportional to the number of polygons stored in any BLACK node, $p = 4$. The term t is time required for merging polygons which are stored in GRAY children nodes of the subtree of OAQ B, which is proportional to the number of GRAY children nodes of subtree of OAQ B. Since each BLACK of OAQ A intersect all BLACK blocks of OAQ B, the created subtree has the same position BLACK nodes as that of the OAQ B. Each GRAY node of the subtree has two BLACK nodes, therefore $t = 2$. The algorithm for merging polygons using method II requires $21 + 8(4 + 2) = 69$ high-level operations.

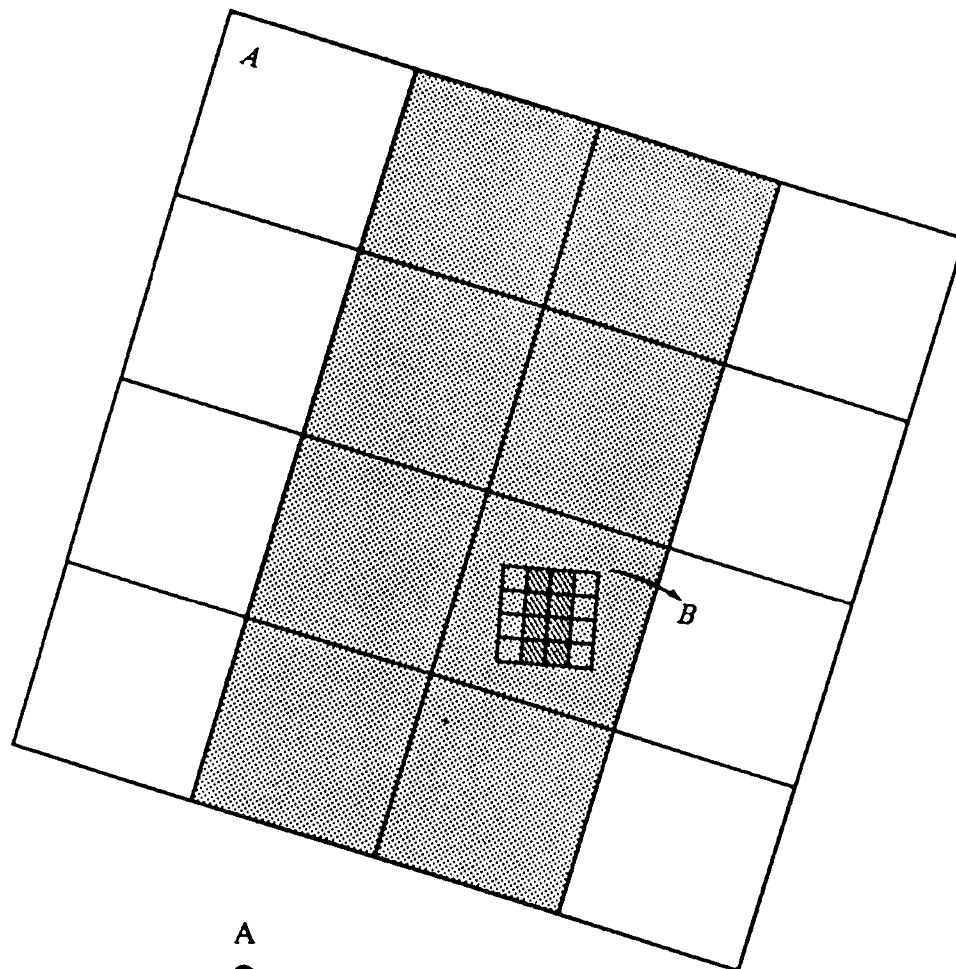
The values discussed above show that the execution time for the intersection OAQ operation is dominated by the execution time for extracting polygons.

4.3.1.4 An example for all blocks of OAQ B inside one block of OAQ A

This subsection discusses another example of analysis of the execution time. Consider Figure 4.11. The figure gives an example for the case of one OAQ of B lies inside the OAQ of A. As mentioned above, the worst-case execution time for polygon extractions $e = (G_A + B_A) \times N_B + W_A + k_1 + k_2$. Since the algorithm requires to know whether pairs of GRAY blocks of OAQ A overlay pairs of GRAY blocks of OAQ B, no action is taken in OAQ B when WHITE node of OAQ A is visited, then the algorithm requires only $O((1 + 3 \times 4) + B_A \times N_B + W_A)$ time.

For the case that all OAQ B is inside one block of OAQ A, then $k_1 = 0$. Since all BLACK blocks of OAQ B lie inside one block of OAQ A, then k_2 is equal to the number of BLACK nodes of OAQ B, $B_B = 8$ (see Figure 4.12). This example shows that only one BLACK node of OAQ A stores polygons, therefore $B_A = 1$. The number of polygons stored in the BLACK node is equal to the number of BLACK node of OAQ. The polygon extraction algorithm requires $(1 + 3 \times 4) + 1 \times 21 + 8 + 8 = 50$ high-level operations.

As mentioned in the first example, the algorithm for merging polygons using method I requires $O(N_A + B_A(s + p) + G_A r)$ time. In this example $s = B_B = 8^2$ and



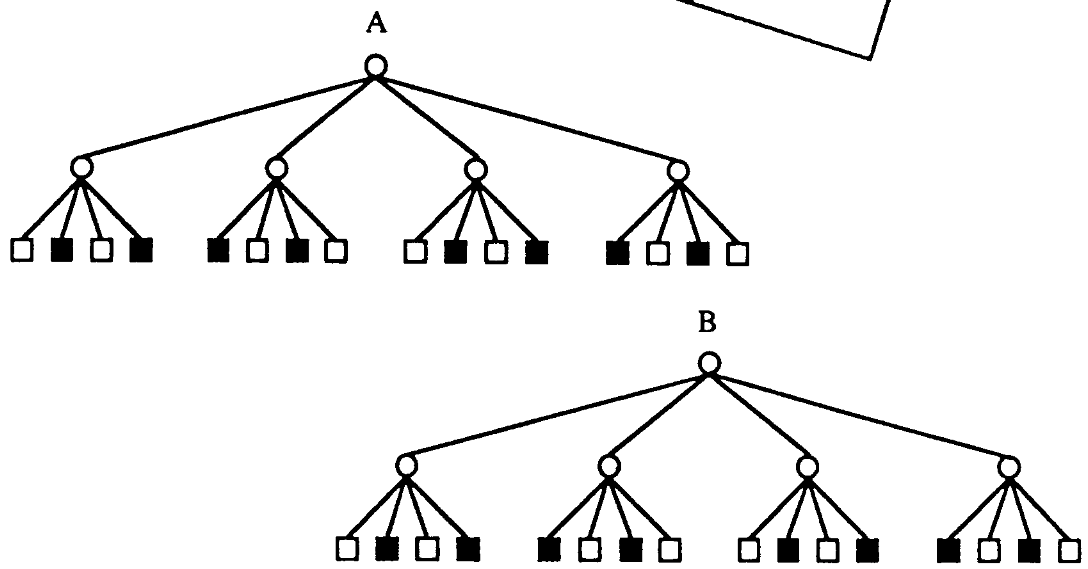


Figure 4.11: Example where the OAQ B lies inside OAQ A.

$p = 8$. In this algorithm, since only one BLACK node of OAQ A stores the polygons, the value of B_A is 1. Since only one GRAY child node stores the merged polygons, the values of G_A and r are 1. The value of $N_A = 5$. The algorithm for merging polygons using method I requires $21 + 1(8^2 + 8) + 1) = 94$ high-level operations.

As discussed in the previous example, the algorithm for merging polygons using method II requires $O(N_A + B_A(p+t) + G_A r)$ time. In this example, $p = B_B = 8$, since all BLACK blocks of OAQ B lie inside one of blocks of OAQ A. The value of $t = 1$, since only one GRAY child node of the subtree of OAQ B exist. The algorithm for merging polygons using method II requires $21 + 1(8+1) + 1 = 31$ high-level operations.

This example shows that the execution time for merging polygons using method I dominates the execution time of the polygon extractions.

4.4 Analysis of union operation

As mentioned in the analysis of intersection, the worst-case execution time of the polygon extractions for intersection is $e = (G_A + B_A) \times N_B + W_A + k_1 + k_2$. The execution time of polygon extractions for union operation is obtained in a similar to the manner described above. Consider the process of examining nodes for performing Boolean union OAQ operation shown in in Figure 3.34. Since the block intersection tests are carried out only for pairs of GRAY blocks and pairs of WHITE blocks OAQ A and BLACK blocks of OAQ B, and no action in OAQ B is taken when BLACK nodes of OAQ A are visited, the process requires $O((G_A + W_A) \times N_B + B_A)$ time.

To extract all polygons resulting from all pairs of intersection between WHITE blocks of OAQ A and BLACK blocks of OAQ B requires $O(u_1)$ time. Extracting all polygons resulting from pairs of WHITE blocks of OAQ A which lie inside the BLACK block of OAQ B or vise versa requires $O(u_2)$ time. The term u_3 is the time required to extract the BLACK blocks of OAQ A. Thus the algorithm for extracting polygons requires $O((G_A + W_A) \times N_B + B_A + u_1 + u_2 + u_3)$ time. The complexity of

merging polygons for the Boolean OAQ union operation is analogous to the execution time of the merging polygons for Boolean intersection operations, except that the value of B_A in equation (4.2) and (4.4) is replaced with W_A depending on which algorithm is used. Therefore the worst-case execution time of merging polygons using method I for the Boolean OAQ union is obtained by substituting W_A to B_A for equation (4.2), which gives

$$O(N_A + W_A(s + p) + G_A r) \tag{4.5}$$

The worst-case execution time for merging polygons using method II is obtained by substituting W_A to B_A for equation (4.4), which gives

$$O(N_A + W_A(p + t) + G_A r) \tag{4.6}$$

The overall time required for Boolean OAQ union operations is $O(e + m)$, where $O(e) = O(G_A + W_A) \times N_B + B_A + u_1 + u_2 + u_3$ is the time required for the polygon extraction algorithm, and $O(m)$ is the time required for merging polygons which is equal to equation (4.5) for method I, and (4.6) for method II. Overall, the worst-case execution time for performing Boolean OAQ operations using method I or II is dominated by the time required for extracting polygons.

Earlier it was mentioned that the best case for process examining nodes for polygon extraction has execution time $\Omega(1)$. The execution time for performing OAQ operation using method I or method II ranges from 1 to $(G_A + B_A) \times N_B + W_A$ for the intersection operation, and from 1 to $(G_A + W_A) \times N_B + B_A$ for the union operation.

OAQ representation allows Boolean OAQ operations between any two regions with different orientation, centroid, and resolution. The worst-case excecution time is closely related to the product of the number of BLACK and WHITE nodes of OAQ A, and the number of nodes of OAQ B. The excecution time is mainly contributed by the execution time of polygon extractions. Once the polygon extractions is complete, then the polygons are merged in where the worst-case execution time is proportional to the number of nodes of OAQ A.

For Boolean intersection operations, the extracted polygons are taken from all pairs of intersecting BLACK blocks representing BLACK nodes from different OAQs. For union operation, extracted polygons are taken from all pairs of intersecting WHITE blocks of OAQ A and BLACK blocks of OAQ B, and all BLACK blocks of OAQ A. These considerations do not involve the BLACK region of B which lies outside the outer block of A (required for the computation of the Boolean OAQ union operation such as shown in Figure 3.35). As a consequence, to include the part of BLACK blocks of OAQ B which lie outside of OAQ A, object space coordinate range of OAQ A needs to be enlarged. OAQ A will be larger, and all BLACK blocks of OAQ B will be inside the OAQ A (see Figure 3.36). The polygon extractions now are carried out to obtain all pairs of intersection of WHITE blocks of OAQ A and BLACK blocks of OAQ B, and all BLACK blocks of OAQ A.

Chapter 5

Conclusions

The work for this thesis examined an alternative for storing the raster representation of an object. As a result, the algorithm for converting connected planar objects to OAQ representation was proposed and tested (chapter 2, and chapter 3), and the approach was implemented. The results of the research have been satisfactory, and the objectives outlined in section 1.6.1 have been met. In general, the accomplishments can be stated as follows :

1. A unique OAQ representation was established which takes into account large eccentricity and aligns the quadtree to the object's principal axis of inertia. It provides a near optimal (in terms of the number of nodes) quadtree encoding of connected planar objects, which are rectangular.

2. It was shown that OAQ representation does not require significant extra time to compute the orientation of the object being represented into an OAQ.

3. It was shown that the number of nodes for the OAQ can be constant for varying image size and orientation, but the number of nodes for the standard quadtree basically doubles for every doubling of the image space coordinate range.

4. Boolean OAQ operations have been designed, implemented and tested for OAQ representations of regions which differ in orientation, centroid, and resolution.

5. It was shown that Boolean OAQ operations requires $O(e+m_k)$ time, where $O(e)$ is the time required for polygon extractions, $O(m_k)$ is the time required for merging polygons, and $k = 1, 2$ are the index for the polygon merging method chosen. In general, the main contribution of time complexity comes from traversing both OAQs and extracting polygons from the intersecting blocks, which is $O(e)$ time. The term e is approximately $N_A \times N_B$, or roughly $O(n^2)$ for n = the number of nodes.

6. The Boolean OAQ operations are designed to produce edge structures representing objects, which can be converted into OAQ representation. In this sense, the algorithms given here are complete.

7. Experimental results show that the saving in the number of nodes ranged from 0% to 19%. For ideal cases, such as the rectangle shown in Figure 4.1, the saving in the number of nodes can be dramatically increased.

5.1 Suggestions for futher work

OAQ representation is one of the approaches for representing regions. It takes advantage of any second order moment of inertia of planar objects by centering the quadtree axes at the object's centroid and aligning the quadtree axes with the object's principal axis of inertia. Boolean OAQ operations can be performed between two OAQs representing two objects which have different resolutions, centroids and orientations. Some other Boolean operations, such as set difference and exclusive or, can be carried out using the methods developed in chapter 3. The set difference of OAQs A and B can be found by extracting the intersection between pairs of BLACK blocks of OAQ A and WHITE blocks of OAQ B. For set difference between OAQ B

and OAQ A, the polygon extractions are carried out when WHITE blocks of OAQ A intersect with BLACK blocks of B. For exclusive or OAQ operation the polygon extractions are carried out when pairs of BLACK blocks of OAQ A and WHITE blocks of B are intersected, and also when BLACK blocks of OAQ B intersect with WHITE blocks of OAQ A.

In computer vision, one of the approaches of determining shape properties is to compute the shape number [2]. The algorithm for making a shape number of a specified order is explained in [2]. Computing the shape number requires placing the chain-code directions to be aligned with the principal axis of inertia. The OAQ representation could be used to advantage since it is already aligned this way. The chain code representation can be directly constructed from the OAQ representation, and then the first difference of the chain code can be computed as well as the shape number. Having the OAQ representation, the orientation and scale independent shape number is easily computed.

The OAQ representation can be extended into Orientation Adaptive Octree Representation which can represent three dimensional objects. The computation of the orientation of the objects can be computed using methods similar to the methods developed in section 2.1 and extended for three dimensional objects.

References

[1] Beer, F.P. and Johnston, E.R. *Mechanics for Engineers: Statics and Dynamics*, 2nd edition, McGraw-Hill, New York, 1962.

[2] Ballard, D.H. and Brown, C.M. *Computer Vision*, Prentice-Hall, Inc, New Jersey, 1982.

[3] Blum, H. "A transformation for extracting new descriptors of shape in". *Models for the perception of speech and visual form*, Wathen-Dunn, W., ed., MIT Press, Cambridge, Mass, 1967.

[4] Gargantini, I. "An effective way to represent quadtree", *Comm. of the ACM*, vol.25, no.12, 1982, pp.905-910.

[5] Gonzales, R.C. and Wintz Paul. *Digital Image Processing*, Addison-Wesley, 1987.

[6] Foley, J.D. and Van Dam, A. *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, 1984.

[7] Forsythe, George E; Malcom, Michael A; and Moler, Cleve B. *Computer Method fot Mathematical Computations*, Prentice-Hall, Inc, New Jersey, 1977.

[8] Freeman, H. "Computer processing of line-drawing images", *Computing Surveys*, vol.6, no.1, 1974, pp.57-97.

[9] Freeman, H. and Shapira, R. "Determining the minimum area encasing rectangle for an arbitrary closed curve", *Comm. of the ACM*, vol.18, no.7, 1975, pp.409-103.

[10] Hegron, Gerard. *Image Synthesis Elementary Algorithms*, MIT Press, Cambridge, 1988.

[11] Hu, M.K. "Visual pattern recognition by moment invariants", *IRE Trans. on Information Theory*, vol.It-8, 1962, 179-187.

[12] Hunter, G.M. "Operations on images using quadtrees", *IEEE Trans. on Pattern Analysis and Machine Intelligence* vol.1, no.2, 1979, pp.148-153.

[13] Kreyszig, E. *Advanced Engineering Mathematics* , 3rd edition, John Willey and Sons, 1972.

[14] Lauzon, J.P. "Two-dimensional run encoding for quadtree representation", *Computer Vision, Graphics, and Image Processing*, vol.44, 1985, pp.87-116.

[15] Merrill, R.D. "Representation of contours and regions for efficient computer search", *Comm. of the ACM*, vol.16, no.2, 1973, pp.69-82.

[16] Nagy, G. "Geographic data processing", *Computing Surveys*, vol.11, no.2, pp.139-181.

[17] Nickerson, B.G. *Automated Cartographic Generalization For Linear Map Features* PhD Dissertation, Rensselaer Polytechnic Institute, Troy, New York, May, 1987.

[18] Nickerson, B.G and Hartati, Sri. "Constructing Orientation Adaptive Quadtrees", *Proceedings Graphics Interface 90* Halifax, May, 1990, pp.190-195.

[19] Rogers, D.F. *Procedural Elements for Computer Graphics*, McGraw-Hill, New York, 1985.

[20] Rogers, D.F. and Adams, J.A. *Mathematical Elements for Computer Graphics*, 2nd edition, McGraw-Hill, New York, 1990.

[21] Samet, H. "Region representation: quadtrees from binary arrays", *Computer Vision, Graphics and Image Processing*, vol.13, 1980, pp.88-93.

[22] Samet, H. "An algorithm for converting raster to quadtrees", *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol.3, no.1, 1981, pp.93-96.

[23] Samet, H. "Neighbor finding techniques for image represented by quadtrees", *Computer Vision and Graphics and Image Processing*, vol.18, 1982, pp.37-57.

[24] Samet, H. "An algorithm for convertion of quadtrees to raster", *Computer Vision, Graphics and Image Processing*, vol.26, 1984, pp.1-16.

[25] Samet, H. "The quadtree and related hierarchical data structure", *Computing Surveys*, vol.16, no.2, 1984, pp.187-259.

[26] Samet, H. *The Design and Analysis of Spatial Data Structures*, Addison - Wesley, 1990.

[27] Samet, H. *The Applications of Spatial Data Structures*, Addison - Wesley, 1990.

[28] Samet, H. and Tamminen, H. "Computing geometric properties of images represented by linear quadtrees", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1985, pp.229-239.

[29] Samet, H. and Shaffer, A.C. "A model for the analysis of neighbor finding in pointer-based quadtrees", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1985, pp.717-720.

[30] Samet, H. and Webber, R.E. "Hierarchical data structure and algorithm for computer graphics. Part I: Fundamental", *Computer Graphics and Applications*, 1988, pp.48-68.

[31] Samet, H. and Webber, R.E. "Hierarchical data structure and algorithm for computer graphics. Part II: Applications", *Computer Graphics and Applications*, 1988, pp.59-75.

[32] Shaffer, C.A. and Samet, H. "Set operations for unaligned linear quadtrees", *Computer Vision, Graphics, and Image Processing*, vol 50 1990, pp.29-49.

[33] Shiflet, Angela B. *Discrete Mathematics for Computer Science*, West Publishing Company, St Paul, 1987.

[34] Weiler, K. "Polygon comparison using a graph representation", *Comm. of the ACM*, vol.21, no.4, 1980, pp.10-18.

VITA

Candidate's full name	: Sri Hartati
Place and date of birth	: Surakarta, Indonesia September 21, 1961
Permanent address	: Jl. Sidoluhur 57, Rt02/Rw03 Lawiyan Surakarta, Central Java, 57148 Indonesia
Schools attended	: Surakarta Yunior High School I Surakarta, Indonesia
1974 - 1976.	
	Surakarta Senior High School I Surakarta, Indonesia
1976 - 1980.	
Universities attended	: Gadjah Mada University Yogyakarta, D.I.Y., Indonesia BSc. (Physics) 1980 - 1986. University of New Brunswick Fredericton, N.B, Canada 1988 - 1990.
Publications	: Nickerson, B.G and Hartati, Sri. "Constructing Orientation Adaptive Quadtrees", *Proceedings Graphics Interface 90* Halifax, May, 1990, pp.190-195.