A Non-Recursive Border Finding Algorithm for Linear Quadtree Based Images

by

Shao Kang Tang

B.A.Sc., The University of Waterloo, 1992

A thesis submitted in partial fulfillment of

the requirements for the degree of

Master of Applied Science

in

The Faculty of Graduate Studies

Department of Electrical Engineering

We accept this thesis as conforming

to the required standard

The University of British Columbia

July 1994

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

(Signature)

Department of ELECTRICAL ENGINEERING

The University of British Columbia
Vancouver, Canada

Date August 2, 1994

DE-6 (2/88)

Abstract

Given a digital binary image represented in a linear quadtree, its border is to be found as another linear quadtree. A new algorithm is proposed that can generate the border in sorted order. A pattern is found for the order of the edge pixels of a node. The differences of the location codes for the edge pixels of a node with grouping factor g is stored in a lookup table. The lookup table also provides the information about which edge a given pixel is on. Through a probabilistic analysis, the new algorithm is able to avoid recursion which reduces the number of neighbour finding operations significantly. The algorithm performs better, when compared to the one by Yang & Lin, in both the run time (40% improvement) and the number of neighbour finding operations incurred (60% reduction).

Contents

List of Tables

List of Figures

Acknowledgement

I am grateful to Dr. Schrack for his guidance and patience, and I am indebted to the financial support from NSERC of Canada. I also wish to express my sincere thanks to my parents, relatives and friends for their support.

Chapter 1

Introduction

1.1 Problem definition

Finding the borders of an image is a well-studied problem in image processing[1]. If black pixels represent the objects in the image and white pixels represent the background, a binary image is obtained. It is a relatively easy task to find borders of a digital binary image because the criterion is simple: a pixel is on the border if one of its neighbours is white. However, there have been few algorithms designed specifically for linear quadtree based images.

A linear quadtree is a variation of the (regular) *quadtree* data structures [Same 90]. Quadtrees are used in areas of computer vision, computer graphics, image processing, spatial databases, cartography, and geographic information systems (GIS). Border finding is a common operation in these applications. For example, the query "find all streets on the border of Vancouver" would require a GIS perform the following operations: find the

[1]It is termed *edge detection* in the image processing literature.

border of a map representing Vancouver and then intersect the result with a map storing the streets of Vancouver.

The quadtree is constructed by dividing a $2^r \times 2^r$ raster domain into four quadrants, where r is the resolution. If a quadrant is only partially contained in an object in the image, that quadrant is subdivided recursively into its four subquadrants. The above process continues until either the subquadrant is completely contained within an object, completely outside all objects, or the pixel level has been reached.

A quadtree is usually implemented as a pointer-based hierarchical data structure where the root node represents the whole raster domain. Each node represents a subquadrant on the raster, and is either black (completely contained within an object), white (completely outside all objects), or grey (partially contained by an object). If a node is grey, then it has four pointers for its four children representing the subquadrants.

Since pointers occupy much space, a variation called *linear quadtree* is introduced that stores only the black nodes in a list or an array [Garg 82]. Each black node is represented by a *location code* and its associated *grouping factor* or *level.* A location code is obtained by traversing the links of the quadtree. Each link is represented by a quaternary digit:

- 0 for the south-west child,
- 1 for the south-east child,
- 2 for the north-west child, and
- 3 for the north-east child.

The location code (*loc*) of a black node is the concatenation of the quaternary

digits on the path from the root to the node itself. If the node is not a pixel, 0's are appended to fill the location code to r digits. The grouping factor (g) represents the size of the node: $g = 0$ means a pixel, $g = 1$ means a 2×2 node, ..., $g = k$ means a $2^k \times 2^k$ node. The level of a node is the resolution minus its corresponding grouping factor; it represents the depth of the node in the quadtree. For example, consider Figure 1.1 , the node consisting of the pixels {48,49,50,51} is represented by the pair (loc,g)=(0300_4,1)=(48_{10},1).

A linear quadtree consists of the set of (loc,g) pairs representing the black nodes of the objects in the image. It is also required that the pairs be sorted in ascending order by the location code.

A useful property of the location code of a pixel is that it contains (x,y) values of the pixel in the Cartesian coordinate system. Specifically, each location code is the interleaving of the bits of the x and y coordinates. For example, the location code $49_{10} = 0301_4$ is 00110001_2 in binary. If the bits are separated alternately, starting with the first bit as the most significant bit of the y coordinate, one obtains $y = 0100_2 = 4$ and $x = 0101_2 = 5$, which are precisely the coordinates of the pixel 49 in Figure 1.1.

The terms 4-connected and 8-connected define the adjacency relationship between any two pixels. If two pixels are adjacent, meaning they are neighbours in one of the four directions east, south, west, or north, then they are 4-connected. If two pixels are adjacent in one of the eight directions east, south, west, north, south-east, south-west, north-west, or north-east, then they are 8-connected. The border pixels are those that have at least a white neighbour, be it 4- or 8-connected; however, in this thesis, only 8-connected border will be considered.

The edge pixels of a node are distinguished from the border pixels. A

1 3 1 11 1 3 1

170	171	174	175	186	187	190	191	234	235	238	239	250	251	254	255
168	169	172	173	184	185	188	189	232	233	236	237	248	249	252	253
162	163	166	167	178	179	182	183	226	227	230	231	242	243	246	247
160	161	164	165	176	177	180	181	224	225	228	229	240	241	244	245
138	139	142	143	154	155	158	159	202	203	206	207	218	219	222	223
136	137	140	141	152	153	156	157	200	201	204	205	216	217	220	221
130	131	134	135	146	147	150	151	194	195	198	199	210	211	214	215
128	129	132	133	144	145	148	149	192	193	196	197	208	209	212	213
42	43	46	47	58	59	62	63	106	107	110	111	122	123	126	127
40	41	44	45	56	57	60	61	104	105	108	109	120	121	124	125
34	35	38	39	50	51	54	55	98	99	102	103	114	115	118	119
32	33	36	37	48	49	52	53	96	97	100	101	112	113	116	117
10	11	14	15	26	27	30	31	74	75	78	79	90	91	94	95
8	9	12	13	24	25	28	29	72	73	76	77	88	89	92	93
2	3	6	7	18	19	22	23	66	67	70	71	82	83	86	87
0	1	4	5	16	17	20	21	64	65	68	69	80	81	84	85

Figure 1.1: Linear quadtree domain of resolution 4

pixel is an edge pixel if it is on an edge of a node. A border pixel must be an edge pixel of a node; an edge pixel need not be a border pixel when all of its neighbours are black.

1.2 Literature review

There have been several definitive algorithms for finding borders of a raster image. For example, the algorithm by Pavlidis [Pavl 82] follows the contour of an object and generates the borders as chain codes. Rosenfeld & Kak [Rose 82] find borders by translating the reverse of the image in four directions and then intersect with the original image. However, the first algorithm specifically designed for quadtree-base images seems to be the one by Dyer et al. [Dyer 80]. It adopts the paradigm of contour following as documented in [Pavl 82] with a few rules designed for quadtrees. Li and Loew [Li 84] also use the idea of contour following but they first convert a quadtree to a raster image before proceeding to find the borders.

Atkinson et al. [Atki 84] and [Atki 85] discovered a completely different approach for finding borders. They achieved the result by repeatedly eliminating internal black nodes. Although the algorithms were presented as linear octree algorithms, they were really simple extensions of the corresponding linear quadtree algorithms.

Dillencourt and Samet [Dill 88] invented yet another paradigm which was based on the concept of active borders. While traversing the nodes of a linear quadtree in order, they kept a set of active borders that define the boundary between the nodes that have been visited and those that have not. Borders of the objects are recorded by checking neighbouring

nodes of the active borders. Qian and Bhattacharya [Qian 89] adapted an algebraic approach to this problem. They converted a quadtree to a picture polynomial which can then be multiplied by another polynomial representing the operation of border finding. Franciosa and Nardelli [Fran 91] used a guaranteed approximation algorithm which was mainly designed for on-line computing a quadtree border. They generated the approximated borders of a quadtree and then successively refined the approximations.

Yang and Lin [Yang 90] returned to the approach of Atkinson et al. [Atki 85] and improved upon it by utilizing a new neighbour finding algorithm and some rules to avoid most of the sorting required. The new algorithm presented in this thesis improves that of Yang and Lin [Yang 90] by avoiding sorting and by removing the need for recursion.

The characteristics of the algorithms discussed above are summarized in Tables 1.1 and 1.2. To make the tables more compact, the following synonyms are introduced.

Paradigm:

cf Contour following

ti Translations and intersections

ab Active borders

alg Algebraic (polynomial) approach

ga Guaranteed approximations

re Repeated elimination

Input:

r Raster

rqt Regular quadtree

rqt2r Regular quadtree converted to raster

lqtw Linear quadtree with white nodes as input as well

rqt2pp Regular quadtree converted to picture polynomial

rqta Regular quadtree with adjacency information embedded

lot Linear octree

lqt Linear quadtree

Output:

cc Chain codes

rp Random pixels

pv Polygonal vertices

pp2rqte Picture polynomial converted to regular quadtree with border enlarged

rqtn Regular quadtree with possibly non-pixel nodes

lot Linear octree

lqt Linear quadtree

Adjacency:

8 8-connectedness

4 4-connectedness

fv8 Face- and vertex-connectedness (corresponding to 8-connectedness in 2D)

Applicability:

eism External and internal borders of singly and multiply connected regions

es External border of a singly connected region

Time complexity:

P Perimeter of the object

N Number of nodes in the tree

G Total number of grey nodes at all levels

M Number of border voxels (or pixels in 2D) in the tree

k Maximum node grouping

n Resolution

As one can see from the tables, the various algorithms assume different input and output conditions, therefore, it is difficult to compare them. The new algorithm is listed in Table 1.2 and is compared to the one by Yang & Lin [Yang 90].

Table 1.1: Characteristics of border finding algorithms

	[Pavl 82]	[Rose 82]	[Dyer 80]	[Li 84]
Paradigm	cf	ti	cf	cf
Input	r	r	rqt	rqt2r
Output	cc	rp	cc	cc
Adjacency	8	8	4	4
Recursive?	no	no	yes	no
Sorting?	no	no	no	no
Neighbour finding?	no	no	yes	no
Applicability	eism	eism	es	es
Complexity	N/A	N/A	O(P) average	N/A

	[Dill 88]	[Qian 89]	[Fran 91]
Paradigm	ab	alg	ga
Input	lqtw	rqt2pp	rqta
Output	pv	pp2rqte	rqtn
Adjacency	N/A	N/A	4
Recursive?	no	no	no
Sorting?	no	no	no
Neighbour finding?	no	no	no
Applicability	eism	N/A	N/A
Complexity	N/A	N/A	O(G) worst

Table 1.2: Characteristics of lqt/lot border finding algorithms

	[Atki 84]	[Atki 85]	[Yang 90]	new
Paradigm	re	re	re	re
Input	lot	lot	lot	lqt
Output	lot	lot	lot	lqt
Adjacency	fv8	fv8	fv8	8
Recursive?	yes	yes	yes	no
Sorting?	no	yes	few	no
Neighbour finding?	yes	yes	yes	yes
Applicability	eism	eism	eism	eism
Complexity	O(kn(N+M))	N/A	O(N)	O(N)
	worst		average	average

Chapter 2

Algorithm Description

There are two novelties in the proposed algorithm:

- The edge pixels of a quadtree node can be generated in sorted order.
- A probabilistic analysis leads to the non-recursive nature of the algorithm.

The above two points will be described in sections 2.2 and 2.3 respectively before the complete algorithm is presented in Section 2.4. However, the overview of the algorithm will ensue first.

2.1 Overview of the algorithm

The new algorithm adapts the paradigm of *repeated elimination.* This paradigm was first proposed by [Atki 84] and [Atki 85]. It facilitates border finding by eliminating *internal* nodes. The method of determining whether a node is internal is neighbour finding. If all four neighbours, i.e. east, south, west, and north neighbours, of the current node are black, then it is

an internal node. Otherwise, the current node is sub-divided into its four children and the elimination process continues recursively to the pixel level where the border of the original quadtree is determined.

The new algorithm, however, is closer in spirit to the one by Yang & Lin [Yang 90]. Their algorithm is described first, with the reasons for the two novelties of the new algorithm indicated.

2.1.1 Algorithm by Yang & Lin

Yang & Lin [Yang 90] noted that there are two major disadvantages in [Atki 85]:

1. The resulting border pixels are not sorted as required by the linear quadtree representation.

2. The input data need to be dispatched into different sub-arrays according to the grouping factor.

To avoid these shortcomings, Yang & Lin formulated a few rules that helped categorize a node into one of three types: *internal*, *PB-quadrant*[1], or *undetermined.*

- An *internal* node is characterized by having neither grey nor white neighbours, that is, all neighbours are black.

- A node is a *PB-quadrant* if it has no grey neighbour but has at least one white neighbour. PB stands for proper border. Some of the edge pixels of a PB-quadrant lie on the border of the original quadtree.

[1]Actually, the term used in [Yang 90] is *PB-octant.* Their paper discusses octree borders; however, it is also applicable to quadtrees.

- A node is *undetermined* if it is neither internal nor a PB-quadrant, that is, it has at least one grey neighbour.

In contrast to [Atki 85] which sub-divides a node recursively to the pixel level for any non-internal node, the algorithm by Yang & Lin only sub-divides the undetermined nodes. Avoiding unnecessary subdivisions reduces the number of neighbour finding operations. They also introduced the notion of *predict-binary search* which, as claimed, reduces the time complexity of a binary search from O(log N) to O(1) on average.

The algorithm can thus proceed by examining each input node in order. If the current node is internal, discard it. If the current node is a PB-quadrant, the edge pixels of this quadrant are examined to output those that are on the border. Otherwise, as the current node is undetermined, sub-divide the node into its four children and continue the process recursively. The proposed algorithm follows basically the same process but avoids the need for recursion. Non-recursion further reduces the number of neighbour finding operations which in many cases are the bottleneck operations in finding the border using repeated elimination. See also the discussion in Section 2.3.

Since the input nodes are examined in sorted order, the border pixels are output in sorted order, consequently, no sorting is required. The only disadvantage is that the border pixels of a PB-quadrant also need to be output in sorted order. Currently, the algorithm of Yang & Lin inserts each border pixel into its sorted position. The approach is in essence an insertion sort on the border pixels of any PB-quadrant. The new algorithm strives to produce the PB-quadrant border pixels in sorted order without sorting, but at the expense of auxiliary data structures and preprocessing. The idea is

discussed in Section 2.2.

2.2 Generation of sorted edge pixels of a node

To output the border pixels of a PB-quadrant in sorted order, it is considered beneficial to be able to first generate the edge pixels of a PB-quadrant in sorted order. After the neighbour types have been determined for the PB-quadrant, i.e. whether the neighbours are black or white, one can in principle traverse the edge pixels in sequence and output as border pixels only those with at least one white neighbour.

The idea used in the new algorithm for generating sorted edge pixels of a PB-quadrant, and thus of any node, is to find a certain pattern for the differences of the location codes of the edge pixels and to use lookup tables. A lookup table *diff_border_LUT[g]* points to an array that contains the differences of the location codes of the edge pixels of a node with grouping factor g. Since the grouping factor can be anywhere in the range $[0, res]$, $res + 1$ such lookup tables are potentially needed.

For example, referring to Figure 1.1, the node (96,2) has the following edge pixels, in sorted order:

```
96 97 98 100 101 103 104 106 107 109 110 111
```

Thus diff_border_LUT[2] points to an array with the following entries:

```
1 1 2 1 2 1 2 1 2 1 1
```

which are precisely the differences in the first sequence.

As another example, diff_border_LUT[3] points to the array:

```
1 1 2 1 3 2 6 1 3 1 2 6 2 1 2 6 2 1 3 1 6 2 3 1 2 1 1
```

which can be used to generate the edge pixels of a node with grouping factor 3, given the location code of its south-west corner pixel. For example, to generate the edge pixels of the node (192,3), the following calculations utilizing diff_border_LUT[3] will find them in the desired order:

```
192
192 + diff_border_LUT[3][0] = 192 + 1 = 193
193 + diff_border_LUT[3][1] = 193 + 1 = 194
194 + diff_border_LUT[3][2] = 194 + 2 = 196
.
.
.
254 + diff_border_LUT[3][26] = 254 + 1 = 255
```

The difference sequences of a given grouping factor are all the same and independent of the south-west corner pixel. Therefore, the difference sequence diff_border_LUT[2] also applies to any node of grouping factor 2, for example, the node (0,2), or (224,2). This is the direct consequence of the fact that location codes of the pixels of a node are offset by the location code of its south-west corner pixel from the node of the same grouping factor but located with 0 as its south-west corner. For example, in Figure 1.1, if the location codes of the pixels of the node (224,2) are all subtracted by 224, then they correspond to the location codes of the pixels of the node (0,2).

Since the resolution of an input quadtree is a variable and since not all grouping factors will always appear in the input quadtree, it is not feasible to pre-calculate the differences for diff_border_LUT[2], diff_border_LUT[3],. . . ,

diff_border_LUT[res][2], and then hard-code them into the program. Therefore, these lookup tables are generated dynamically.

To facilitate the dynamic generation of the difference LUT's, a pattern is sought. The above examples of difference sequences on first glance appear to consist of random sequences; however, further investigation leads to a pattern. To see the pattern more clearly, a node of grouping factor 4 will be analyzed. The edge pixels are separated into a lower portion and an upper portion which will be considered separately.

2.2.1 Difference LUT for the lower edge pixels of a node

The lower edge pixels of the node (0,4) will be listed together with their differences in Figure 2.1. The symbols SW, S1 etc. will be discussed in Section 2.2.3. The edge pixels are separated into various lines and are called the *e-lines*. The differences are also separated into various lines and are called the *d-lines*. The number at the end of each d-line is the difference between the last number of the preceding e-line and the first number of the following e-line. For example, the number 6 of the d-line (d3) is 16 minus 10, where 10 is the last number of the e-line (e3) and 16 is the first number of the e-line (e4). The other numbers of the d-lines are the differences of the numbers of the preceding e-lines. For example, consider e-line (e3) and d-line (d3): $5-4=1, 8-5=3, 10-8=2$.

The d-line (d1) always contains the number 1.

For the d-lines (d2) to (d5), the following pattern emerges. The numbers

[2]Note that for nodes of grouping factors 0 and 1, the differences of their edge pixels are trivial. Thus, these differences will not be stored but the program using the lookup tables has to handle these two cases separately.

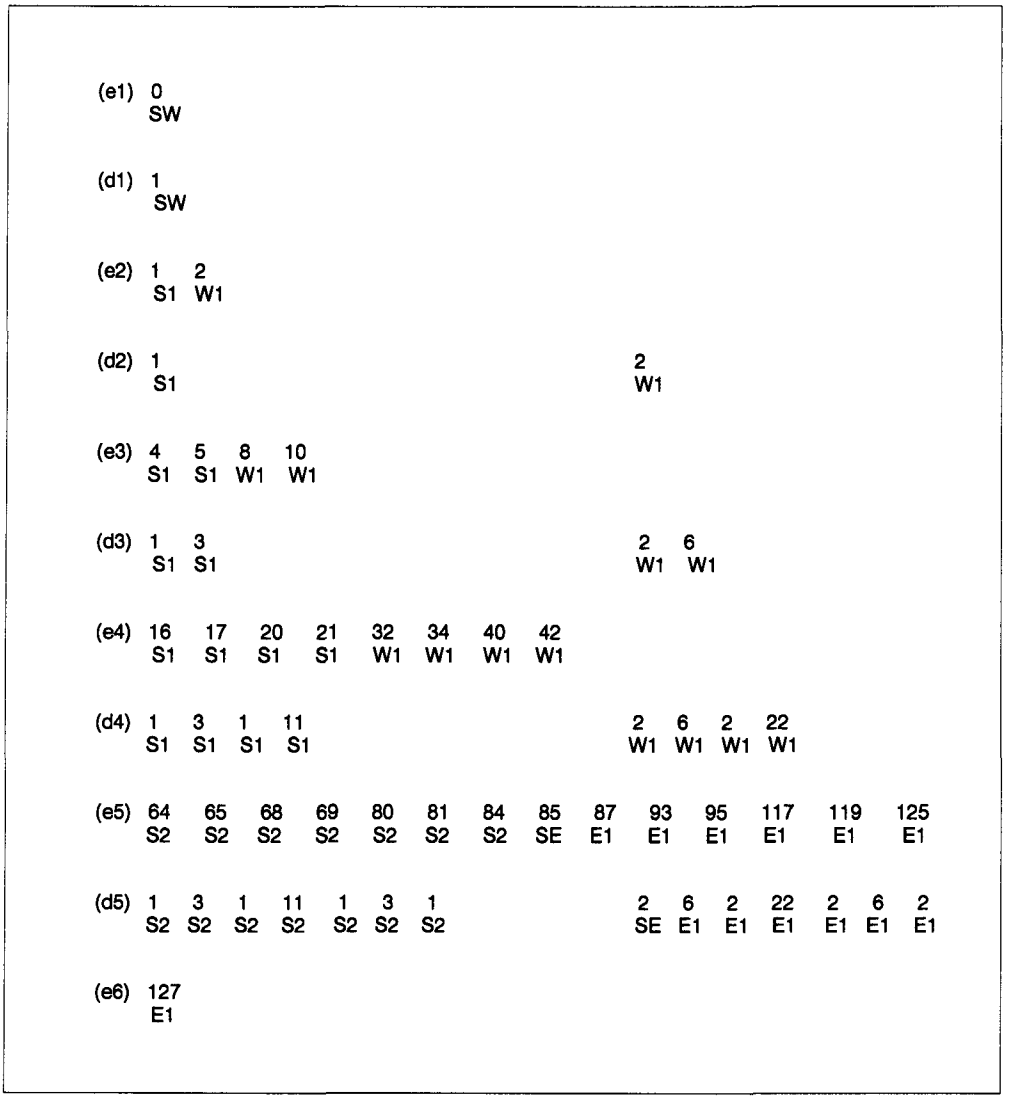

Figure 2.1: Lookup table for lower edge pixels of a node of $g = 4$

on the d-lines are divided into two groups: the *h-group* and the *v-group*. The h-group consists of the numbers in the first half of a d-line; the v-group consists of those in the second half. Each number in the v-group is exactly twice that of the corresponding h-group on any particular d-line. For example on (d4), 2 6 2 22 is twice as much as 1 3 1 11 respectively.

The h-groups of d-lines consist of the same set of numbers as the differences of the location codes across the horizontal axis of the domain. The numbers across the top of Figure 1.1 indicate the differences[3]. The d-line (d_i) has 2^{i-2} entries in its h-group, except the last d-line which has $2^{g-1}-1$ entries. For example,

- (d2) has $2^{2-2} = 2^0 = 1$ entry, the number 1;
- (d3) has $2^{3-2} = 2^1 = 2$ entries : 1 3;
- (d4) has $2^{4-2} = 2^2 = 4$ entries : 1 3 1 11;
- (d5) has $2^{g-1} - 1 = 2^{4-1} - 1 = 7$ entries : 1 3 1 11 1 3 1

in the h-group, where the entries, 1, 1 3, 1 3 1 11, or 1 3 1 11 1 3 1, are the numbers across the top of Figure 1.1.

There are hence

$$1 + 2 \times [2^0 + 2^1 + ... + 2^{g-2} + (2^{g-1} - 1)] = 2^{g+1} - 3 \qquad (2.1)$$

number of entries in the lower portion of diff_border_LUT[g], where the 1 at the beginning accounts for (d1), the summation inside [...] accounts for the h-group on the rest of the d-lines, and the multiplication by 2 accounts for the v-group.

[3]The numbers in the v-group consist of the differences of location codes along the vertical axis of the domain. Thus the designations *v* and *h*.

The above observation can be generalized to a node of any grouping factor in the range $[2, res]$. Thus to generate diff_border_LUT for lower edge pixels of the nodes, $2^{res-1} - 1$ number of differences of the location codes across the horizontal axis are calculated first. Then the above observations are used to generate each d-line in turn. The concatenation of the d-lines gives the differences for the lower edge pixels of the current node.

2.2.2 Difference LUT for the upper edge pixels of a node

Now consider the upper edge pixels of the node (0,4). The edge pixels and their differences are listed in Figure 2.2 and in a format similar to the one in the previous section; however, the differences are now called the *d'-lines.* The number at the beginning of each d'-line is the difference between the last number of the second preceding e-line and the first number of the preceding e-line. For example, the number 22 of the d'-line (d'9) is 213 minus 191, where 191 is the last number of the e-line (e8) and 213 is the first number of the e-line (e9). The other numbers of the d'-lines are the differences of the numbers of the preceding e-lines. For example, consider e-line (e9) and d'-line (d'9): $215 - 213 = 2, 221 - 215 = 6, 223 - 221 = 2$, $234 - 223 = 11, 235 - 234 = 1, 238 - 235 = 3, 239 - 238 = 1$.

It is noticed that the d'-lines are the reverse of the d-lines. For example, (d'10) consists of 6 2 3 1, which is the reverse of 1 3 2 6, the sequence of (d3). Similarly for (d'9) and (d4), or (d'8) and (d5) etc. Therefore, to generate the differences for the upper edge pixels, i.e. the d'-lines, simply generate those of the lower edge pixels, i.e. the d-lines, and reverse the entries.

The difference between the last location code of the lower edge pixels and the first location code of the upper edge pixels is always one. Thus, to

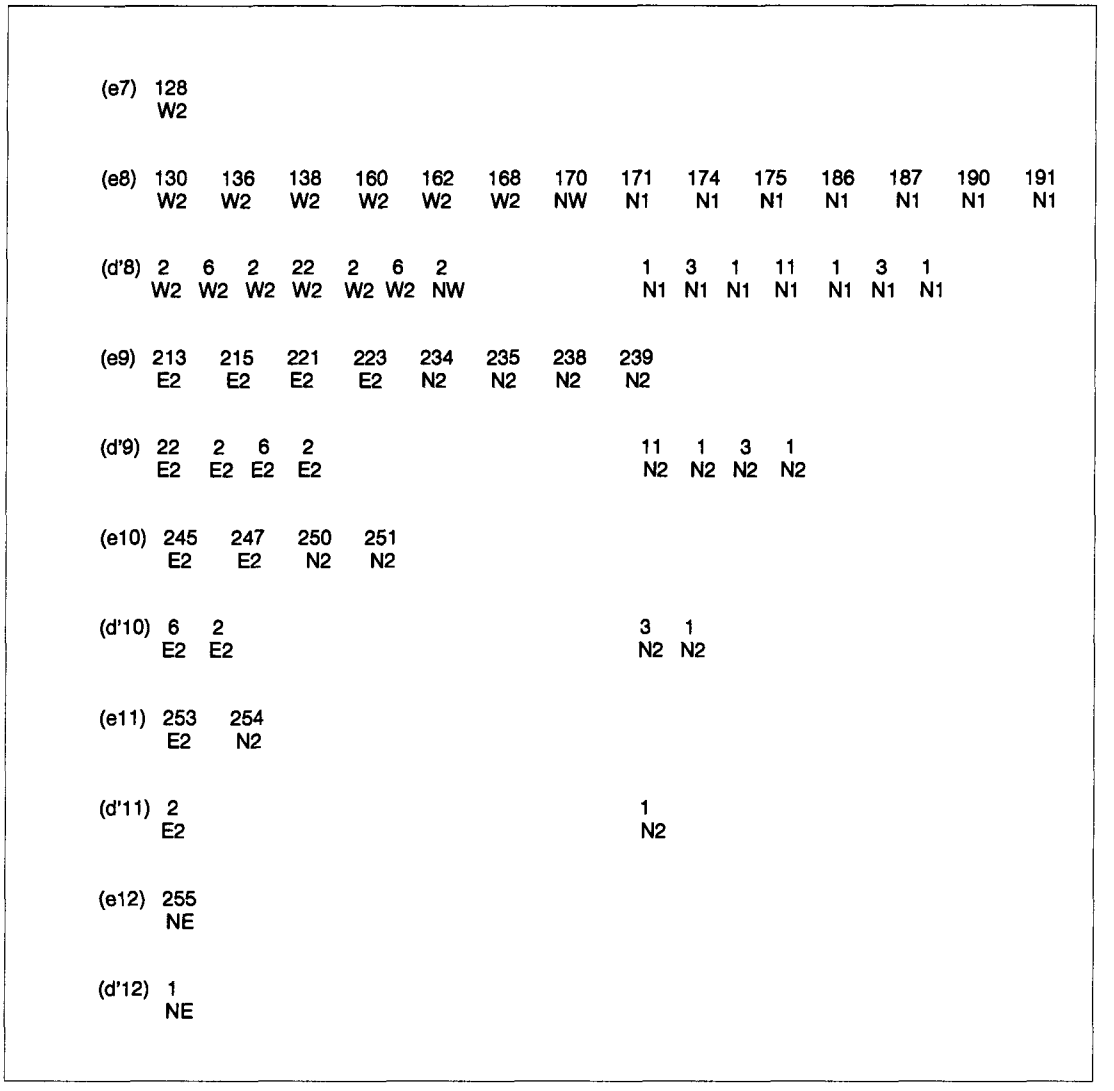

Figure 2.2: Lookup table for upper edge pixels of a node of $g = 4$

get the number 128 on (e7), simply add 1 to the number 127 on (e6).

From the above discussions, diff_border_LUT[4] can be generated by concatenating (d1) to (d5), the number 1, and then (d'8) to (d'12). The number of entries of diff_border_LUT[g] is thus, from Equation 2.1, $2 \times (2^{g+1} - 3) + 1$, or $2^{g+2} - 5$.

2.2.3 Incorporating side information into the difference LUT

Now that the edge pixels of a node can be generated in sorted order, how can one determine whether these edge pixels are on the border of the original input quadtree? A natural idea is to visit each edge pixel and check its neighbours. If one of the edge pixel's four neighbours is white, then that edge pixel belongs to the border of the quadtree. However, there is no need to check all four but only at most two neighbours for each of these edge pixels. For example, if the edge pixel visited is at the north-west corner of the current node, then it is guaranteed that its east and south neighbours are black because these neighbours belong to the current node. Therefore, it is only necessary to check the north and west neighbours of an edge pixel at the north-west corner of the current node. Similarly, it is only necessary to check

- the north and east neighbours of an edge pixel at the north-east corner of the current node;
- the south and east neighbours of an edge pixel at the south-east corner of the current node;
- the south and west neighbours of an edge pixel at the south-west corner of the current node.

For the edge pixels not at the corners of the current node, only one neighbour check is needed. For example, if the edge pixel is on the south side of the current node, then it is guaranteed that its east, west, and north neighbours are black because the east and west neighbours are also on the edge of the current node while the north neighbour is internal to the current node. Therefore, it is only necessary to check the south neighbour of an edge pixel on the south side of the current node. Similarly, it is only necessary to check

- the east neighbour of an edge pixel on the east side of the current node;
- the north neighbour of an edge pixel on the north side of the current node;
- the west neighbour of an edge pixel on the west side of the current node.

Since diff_border_LUT is used for generating the edge pixels of a node, it is convenient that the LUT's will also store the above side information so that appropriate neighbour finding operations can be performed. Due to reasons that will be explained in Section 2.3, the side information will be divided into twelve directions:

```
#define SW      1
#define S1      2
#define W1      3
#define S2      4
#define SE      5
```

```
#define E1      6
#define W2      7
#define NW      8
#define N1      9
#define E2      10
#define N2      11
#define NE      12
```

instead of the usual eight directions: E (east), S (south), W (west), N (north), SE (south-east), SW (south-west), NW (north-west), NE (north-east).

From Figure 2.1, the pattern for the side information can be deduced. For example, there are one S1 on (d2), two S1 on (d3), four S1 on (d4), and seven S2 on (d5). The one, two, four, and seven are exactly the number of entries in each h-group. Therefore, it is possible to store the side information into the lower portion of diff_border_LUT as well. By examining Figure 2.2, it is noticed that the directions of the upper portion of diff_border_LUT are opposite to the directions of the lower portion. For example, NE of (d'12) is the opposite of SW of (d1); N2 of (d'9) is the opposite of S1 of (d4). Therefore, by defining the directions to be the numbers as shown above, the side information of the upper diff_border_LUT can be generated by thirteen minus the corresponding number representing the direction in the lower portion of diff_border_LUT. For example, $13 - SW = 13 - 1 = 12 = NE$ as expected.

2.3 Probabilistic analysis leading to non-recursion

As neighbour finding is usually the bottleneck operation in border finding algorithms using repeated elimination, an effort was made to reduce the number of such operations.

Examining the edge pixels of the input nodes and finding one or two neighbours for each of the pixels may not be efficient in certain cases. For example, consider node P in Figure 2.3,

- finding one neighbour will suffice instead of finding eight neighbours in the south direction;
- finding two neighbours will suffice instead of finding eight neighbours in the east direction.

Since there is no prior knowledge on the size of the neighbours, one possibility is to use recursion as suggested by [Yang 90]. For example, finding one neighbour in the east direction shows that the east neighbour is grey, therefore, the node is subdivided and two neighbour finding operations are performed on neighbours of the size one fourth of the original. The above process continues recursively until the neighbours are no longer grey.

Of course, for recursive subdivisions, neighbours must be found in all four directions. Hence, to resolve the east neighbour for the example in Figure 2.3, $1 + 4 \times 2 = 9$ potential neighbours must be found, where the factor 4 accounts for checking all four neighbours of the south-east and north-east children of the original node. The naive method, on the other hand, only requires 8 neighbour finding operations.

Since neither checking directly all the edge pixels nor using recursion is satisfactory, a hybrid algorithm is devised that incorporates the two. The

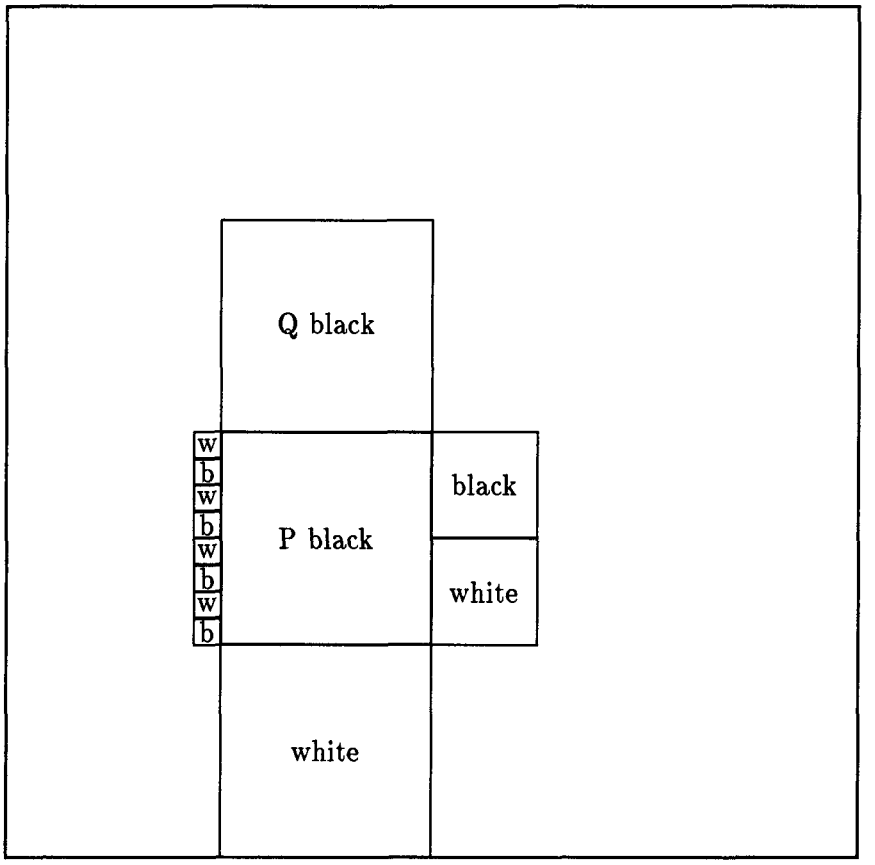

Figure 2.3: A sample node P and its neighbours

hybrid method always finds one neighbour in each of the four directions first. If some of the neighbours found are grey, the algorithm will wait until a certain number of recursive subdivisions are performed before *possibly* checking all edge pixels.

The question is: how many recursive subdivisions should be performed, zero, one, or maybe more? A break-even point in the number of recursive subdivisions is therefore sought that will make the hybrid method a viable approach.

A probabilistic analysis will be used to help determine the break-even point. To simplify the analysis, only the neighbour in one particular direction will be considered.

Let g denote the grouping factor. Consider a node with $g = k$. Let the probabilities of its neighbour being of size $g = k-1$ be p_{k-1}, of size $g = k-2$ be p_{k-2}, and so on, to p_0 for $g = 0$, i.e. the pixel level. Neighbours with $g \geq k$ need not be considered because the neighbour finding operation that is always performed first will resolve such cases.

Let the number of neighbour finding operations required to resolve the type, i.e. black or white, of a neighbour with $g = l, l < k$, using only recursive subdivisions, be f_l. Then

$$f_l = 2^1 + 2^2 + ... + 2^{k-l} = 2^{k-l+1} - 2 \tag{2.2}$$

The factor 4 accounting for the four neighbours of each child node is not included because only a particular direction is considered.

Let E_1 be the expected value of the number of neighbour finding operations needed to resolve the type of a neighbour, using recursive subdivisions

only. Then

$$E_1 = \sum_{i=0}^{k-1} p_i f_i$$

Let E_2 denote the expected value of the number of neighbour finding operations needed to resolve the type of a neighbour using the hybrid method. Let m be the break-even point of the number of recursive subdivisions, then the hybrid method will always subdivide m times before possibly checking the 2^k edge pixels. Then

$$E_2 = \sum_{i=k-m}^{k-1} p_i f_i + (f_{k-m} + 2^k) \sum_{i=0}^{k-m-1} p_i$$

The first term of E_2 is for the case where the grouping factor g of the neighbour is such that $k - m \leq g \leq k - 1$, i.e. the cases that are taken care of by the m subdivisions. The second term of E_2 accounts for the case where the g of the neighbour is such that $0 \leq g \leq k - m - 1$. Then, after f_{k-m} neighbour finding operations have been carried out by means of m subdivisions, the neighbour is still not resolved, hence additional 2^k neighbour finding operations for the edge pixels are included. This occurs with a probability $\sum_{k=0}^{k-m-1} p_i$.

Substituting Equation 2.2 into E_2 and after some algebra, the following relationship is obtained.

$$E_2 = E_1 + 2^{m+1} \sum_{i=0}^{k-m-1} p_i (2^{k-m-1} - \sum_{j=0}^{k-m-1-i} 2^j)$$

It is required that $E_2 \leq E_1$ for the hybrid method to achieve better, or at least the same, efficiency on average. That is,

$$E_2 - E_1 = 2^{m+1} \sum_{i=0}^{k-m-1} p_i (2^{k-m-1} - \sum_{j=0}^{k-m-1-i} 2^j) \leq 0$$

or

$$\sum_{i=0}^{k-m-1} p_i(2^{k-m-1} - \sum_{j=0}^{k-m-1-i} 2^j) \leq 0 \tag{2.3}$$

Provided the probabilities are known, Equation 2.3 allows to solve for the break-even point m. A pragmatic approach is used to provide the probability values. Four typical input quadtrees are processed which give a first approximation for the probabilities p_i. These sample quadtrees are shown in Figures 2.4 to 2.9 where the edge pixels of each node are displayed[4].

The statistics for the sizes of the nodes are summarized in Table 2.1 together with the percentages of occurrences of each node size. The percentages will be used as the approximated p_i in each tree.

Comparing Figures 2.4 to 2.9 with Table 2.1, it is intuitively obvious that half of the input nodes are of size $g = 0$, i.e. pixels. The pixels are primarily distributed along the border of the quadtrees.

Now with the percentage of $g = i$ in Table 2.1 as p_i, Equation 2.3 is examined to find a suitable m.

Consider Figure 2.4, the input quadtree with the shape of a large **F**.

For $k = 0$, the obvious choice is $m = 0$, that is, no subdivision is necessary for pixel level nodes, Equation 2.3 does not apply.

For $k = 1$, the left hand side of Equation 2.3, i.e. the effective $E_2 - E1$, becomes

$$p_0(2^0 - \sum_{j=0}^{0} 2^j) = p_0(2^0 - 2^0) = 0 \tag{2.4}$$

which satisfies the inequality. A node of grouping factor 1 is a 2×2 node where a simple subdivision results in the same number of neighbour finding

[4] A utility program was written for this purpose. It used the algorithm developed in Section 2.2.

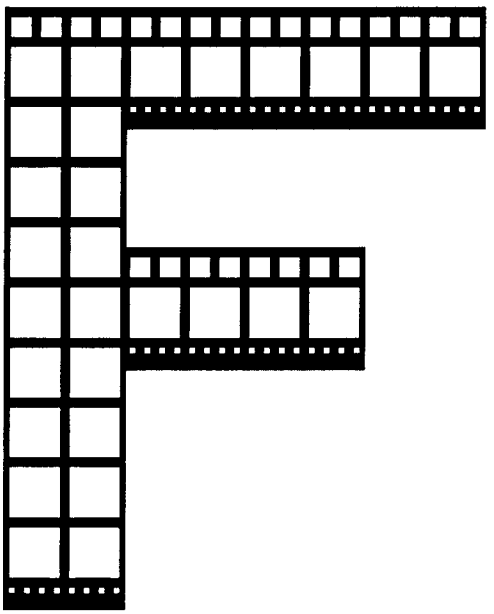

Figure 2.4: Quadtree **F** with edge pixels shown

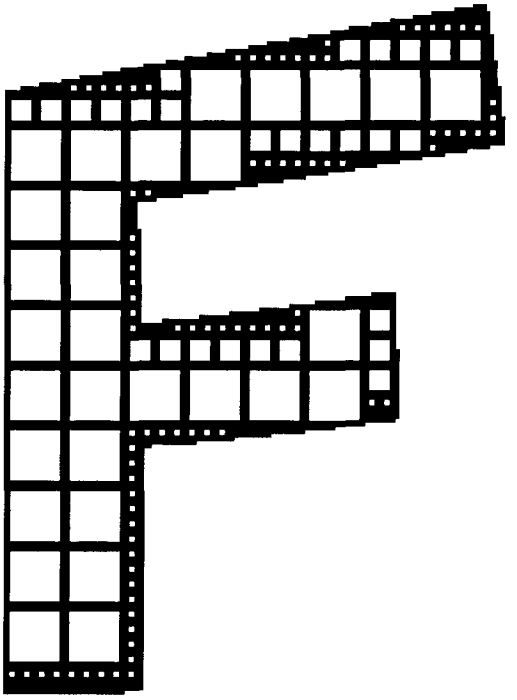

Figure 2.5: Quadtree **slanfF** with edge pixels shown

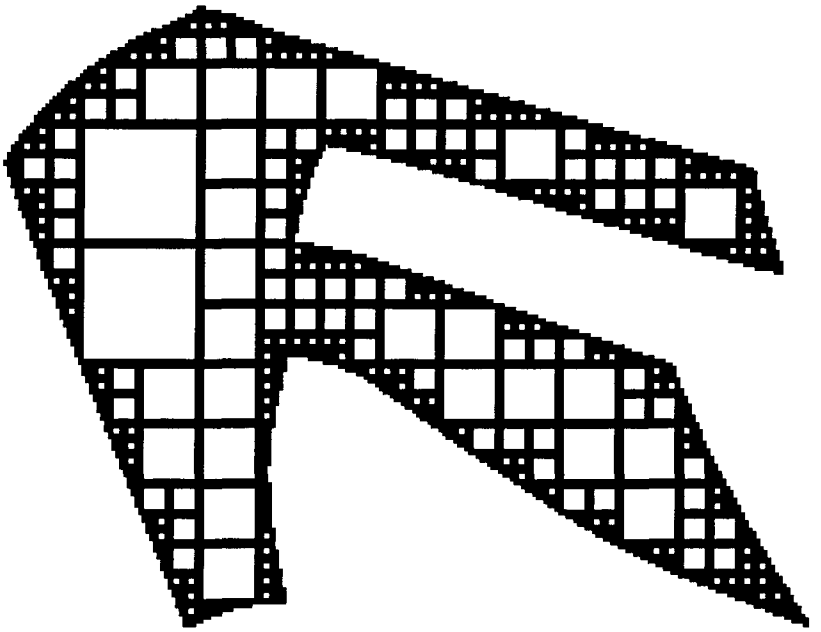

Figure 2.6: Quadtree **distortF** with edge pixels shown

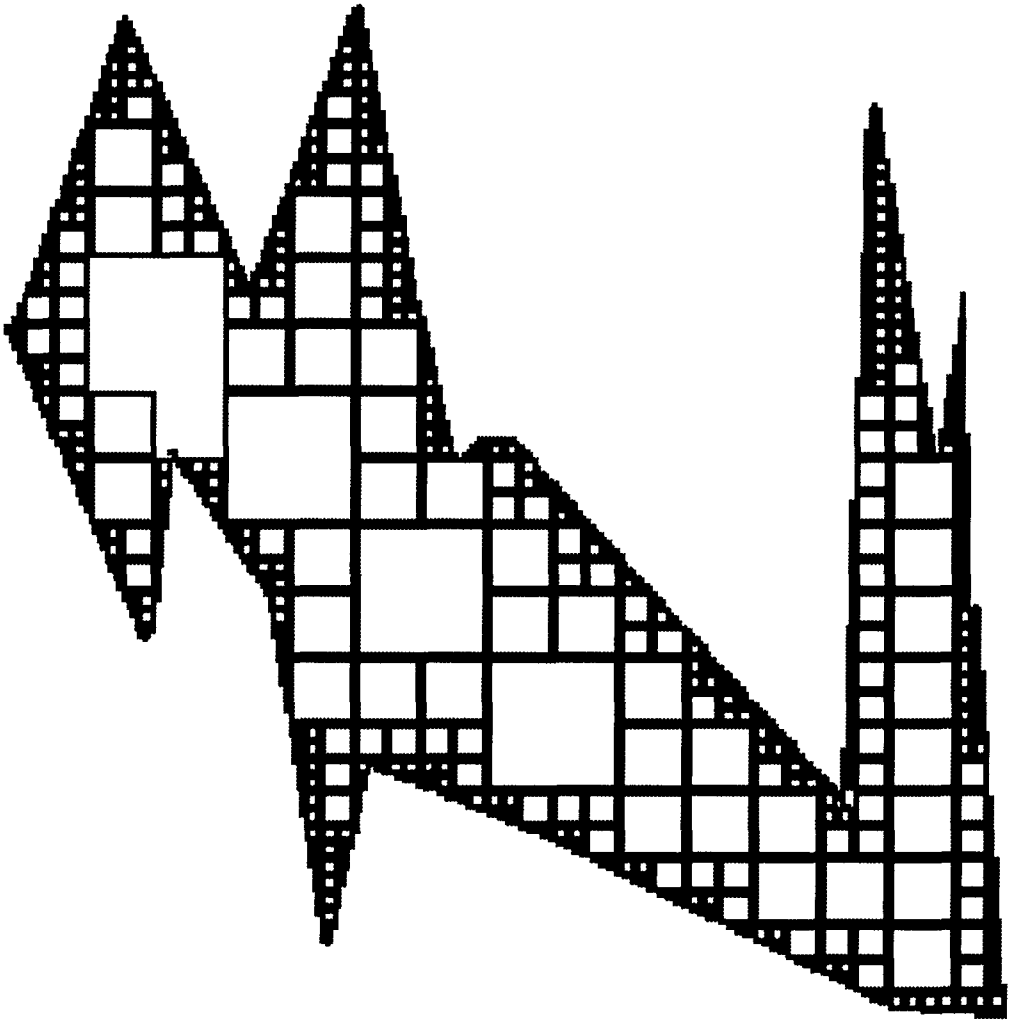

Figure 2.7: Quadtree **random1** with edge pixels shown

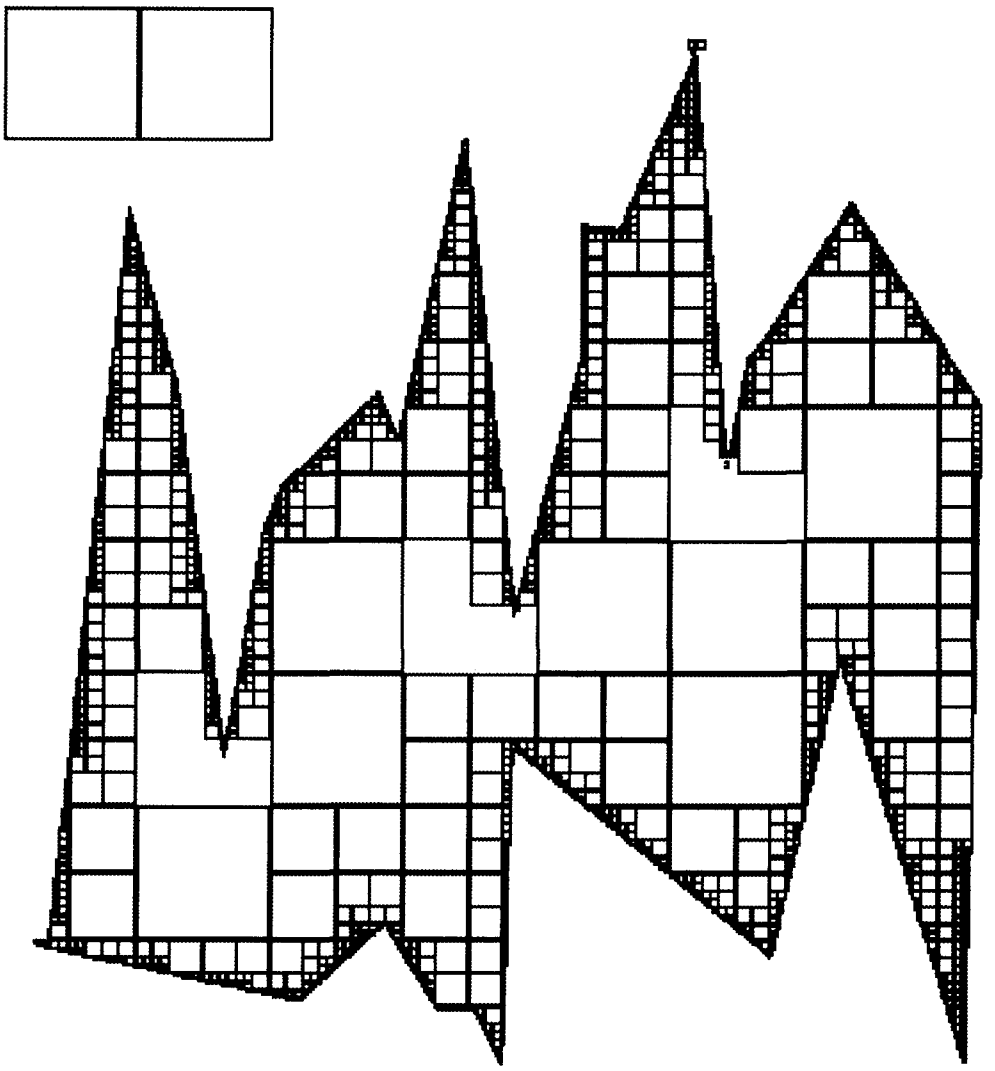

Figure 2.8: Quadtree **random2** with edge pixels shown

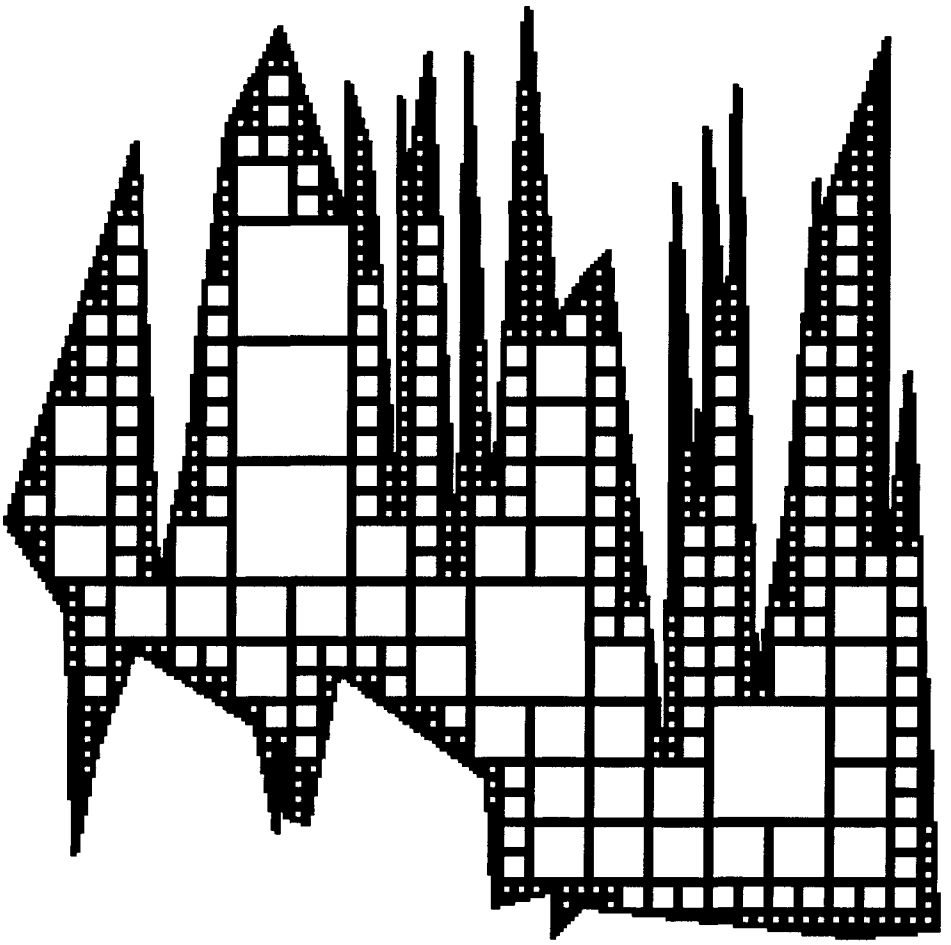

Figure 2.9: Quadtree **random3** with edge pixels shown

Table 2.1: Node size statistics for sample quadtrees

g	**F**		**slantF**		**distortF**	
	# of nodes	%	# of nodes	%	# of nodes	%
8	0	0	0	0	0	0
7	0	0	0	0	0	0
6	0	0	0	0	0	0
5	0	0	0	0	2	0.2
4	28	4.8	30	4.8	24	2.0
3	24	4.1	27	4.3	69	5.9
2	48	8.3	87	13.7	152	13.0
1	96	16.6	135	21.3	309	26.3
0	384	66.2	354	55.9	618	52.6

g	**random1**		**random2**		**random3**	
	# of nodes	%	# of nodes	%	# of nodes	%
10			0	0		
9			0	0		
8	0	0	0	0	0	0
7	0	0	9	0.1	0	0
6	0	0	30	0.4	0	0
5	3	0.2	89	1.2	5	0.2
4	36	2.3	181	2.4	37	1.2
3	82	5.3	438	5.9	155	5.0
2	174	11.3	957	13.0	338	10.9
1	429	27.8	1836	24.9	949	30.6
0	821	53.1	3845	52.1	1613	52.1

operations as that of checking all edge pixels, hence there is no need to perform subdivision, i.e. choose $m = 0$.

For $k = 2$, the left hand side of Equation 2.3 becomes

$$\sum_{i=0}^{1-m} p_i(2^{1-m} - \sum_{j=0}^{1-m-i} 2^j)$$

If $m = 1$, then the above expression becomes Equation 2.4 and results in 0. If $m = 0$, then the above expression becomes

$$p_0(2^1 - 2^0 - 2^1) + p_1(2^1 - 2^0) = p_1 - p_0 \tag{2.5}$$

Substituting the percentages from Table 2.1, $p_1 - p_0 = 0.166 - 0.662 = -0.496$. Both $m = 1$ and $m = 0$ satisfy the inequality of Equation 2.3.

For $k = 3$, the left hand side of Equation 2.3 becomes

$$\sum_{i=0}^{2-m} p_i(2^{2-m} - \sum_{j=0}^{2-m-i} 2^j)$$

If $m = 2$, then the above expression becomes Equation 2.4 and results in 0. If $m = 1$, then the above expression becomes Equation 2.5 and results in -0.496. If $m = 0$, then the above expression becomes

$$p_0(2^2 - 2^0 - 2^1 - 2^2) + p_1(2^2 - 2^0 - 2^1) + p_2(2^2 - 2^0) = 3p_2 + p_1 - 3p_0 \tag{2.6}$$

Substituting the percentages from Table 2.1, $3p_2 + p_1 - 3p_0 = 3 \times 0.083 + 0.166 - 3 \times 0.662 = -1.571$. All three cases $m = 2, m = 1, m = 0$ satisfy the inequality of Equation 2.3.

For $k = 4$, the left hand side of Equation 2.3 becomes

$$\sum_{i=0}^{3-m} p_i(2^{3-m} - \sum_{j=0}^{3-m-i} 2^j)$$

If $m = 3$, then the above expression becomes Equation 2.4 and results in 0. If $m = 2$, then the above expression becomes Equation 2.5 and results in -0.496. If $m = 1$, then the above expression becomes Equation 2.6 and results in -1.571. If $m = 0$, then the above expression becomes

$$p_0(2^3-2^0-2^1-2^2-2^3)+p_1(2^3-2^0-2^1-2^2)+p_2(2^3-2^0-2^1)+p_3(2^3-2^0)$$

which, after simplification, is

$$7p_3 + 5p_2 + p_1 - 7p_0 \tag{2.7}$$

Substituting the percentages from Table 2.1, $7p_3 + 5p_2 + p_1 - 7p_0 = 7 \times 0.041 + 5 \times 0.083 + 0.166 - 7 \times 0.662 = -3.766$. All four cases $m = 3, m = 2, m = 1, m = 0$ satisfy the inequality of Equation 2.3.

Similar calculations were carried out for the other three sample quadtrees. The results of the left hand side of Equation 2.3, i.e. the effective $E_2 - E_1$, are summarized in Table 2.2.

From Table 2.2, it is seen that the effective $E_2 - E_1$ are non-positive in all cases, that is, Equation 2.3 is satisfied for $0 \leq m \leq 4$. Therefore, any integer value in $[0, 4]$ can be chosen as the break-even point m.

Theoretically, the value of m that yields the most negative effective $E_2 - E_1$, i.e. $m = 0$ in Table 2.2, should be chosen because it represents the greatest margin of improvement of the hybrid method over the recursive method. If $m = 0$ is chosen as the break-even point, no recursive subdivisions are needed; consequently it is sufficient to check the neighbours of the edge pixels only.

However, examining the edge pixels of each node without taking advantage of the hierarchical nature of the quadtree structure is considered a naive

Table 2.2: The effective $E_2 - E_1$ using data from sample quadtrees

node of g=k	possible m	Effective $E_2 - E_1$					
		F	**slantF**	**distortF**	**random1**	**random2**	**random3**
1	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0
	0	−0.496	−0.346	−0.263	−0.254	−0.272	−0.214
3	2	0	0	0	0	0	0
	1	−0.496	−0.346	−0.263	−0.254	−0.272	−0.214
	0	−1.571	−1.052	−0.928	−0.979	−0.925	−0.928
4	3	0	0	0	0	0	0
	2	−0.496	−0.346	−0.263	−0.254	−0.272	−0.214
	1	−1.571	−1.052	−0.928	−0.979	−0.925	−0.928
	0	−3.766	−2.715	−2.363	−2.507	−2.333	−2.444
5	4			0	0	0	0
	3			−0.263	−0.254	−0.272	−0.214
	2			−0.928	−0.979	−0.925	−0.928
	1			−2.363	−2.507	−2.333	−2.444
	0			−5.397	−5.640	−5.257	−5.695
6	5					0	
	4					−0.272	
	3					−0.925	
	2					−2.333	
	1					−5.257	
	0					−11.1218	
7	6					0	
	5					−0.272	
	4					−0.925	
	3					−2.333	
	2					−5.257	
	1					−11.122	
	0					−22.980	

or brute-force approach. It is therefore decided that $m = 1$ be chosen for the hybrid algorithm with the $m = 0$ case as a possibility for future research.

Choosing $m = 1$ for the present investigation involves the following rationales. First of all, the percentages from Table 2.1 merely approximate the p_i's. The percentages for $g = i$ in Table 2.1 only indicate the probability of finding a node of grouping factor i if a random position of the image is picked. On the other hand, p_i is the probability of finding a neighbour of grouping factor i for a node with $g > i$. Considering locations of the nodes within an image is important in estimating the p_i's. Secondly, as can be seen from Figures 2.4 to 2.9, frequently nodes have neighbours of equal size or quarter of the size. For neighbours of equal size, the first neighbour finding operation that is always performed will take care of the case. For neighbours quarter of the size, one recursive subdivision, which corresponds to $m = 1$, suffices. Lastly, Table 2.2 shows that $m = 1$ yields the second most negative effective $E_2 - E_1$, only behind the $m = 0$ case.

Since $m = 1$ is chosen, no subdivision is really necessary. With a specially designed data structure, recursion can be eliminated altogether. Besides the four corners SW (south-west corner), SE (south-east corner), NE (north-east corner), and NW (west-west corner), the edge pixels of a node are divided into eight sections: S1 (part 1 of south edge), S2 (part 2 of south edge), E1 (part 1 of east edge), E2 (part 2 of east edge), N1 (part 1 of north edge), N2 (part 2 of north edge), W1 (part 1 of west edge), W2 (part 2 of west edge), as shown in Figure 2.10.

The neighbour types of a node for each of these eight edge sections are stored in an auxiliary data structure. For example, referring to the node P in Figure 2.3,

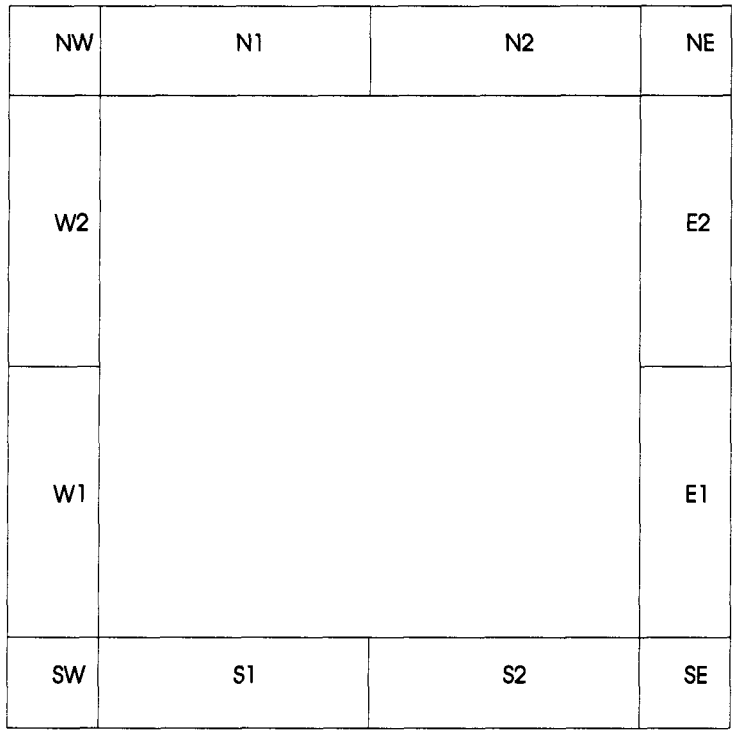

Figure 2.10: Sections of the edge of a node

- both S1 and S2 have white neighbours;
- both N1 and N2 have black neighbours;
- E1 has a white neighbour and E2 has a black neighbour;
- both W1 and W2 have grey neighbours.

A particular corner of a node inherits the neighbour types from those of the adjacent edge sections. For example, the types of the north and west neighbours of the corner NW are the same as the neighbour types of N1 and W2 respectively.

A bit vector, named *snt*, short for *sub-neighbour type*, is used as the auxiliary data structure for the purpose of storing the neighbour type information. Two bits are allocated for each of the eight sections, thus an unsigned 16-bit integer is sufficient for storage. The binary values assigned for the neighbour types are:

- 00 for unset;
- 01 for white;
- 10 for grey;
- 11 for black.

Figure 2.11 shows *snt* for the node P in Figure 2.3.

2.4 The complete algorithm

After the ideas of an ordered generation of the edge pixels and non-recursion have been explained in the previous sections, the complete new algorithm

N2	N1	W2	W1	S2	S1	E2	E1
11	11	10	10	01	01	11	01

where 00 -- unset
01 -- white
10 -- grey
11 -- black

Figure 2.11: The *snt* vector of the node P in Figure 2.3

will now be described. Pseudocodes and/or descriptions for the various routines of the new algorithm are presented. The C codes are listed in Appendix A.

2.4.1 The main routines

The **main()** routine performs the disk I/O. Its major role is to invoke the routine **BorderFind()** for each of the input nodes and to initialize all *snt* bit vectors to zero.

BorderFind():

```
/* Purpose: find the border pixels of the current node */

FOR the four directions of current node DO
   IF snt is unset THEN
      CALL Setsnt()
   ENDIF
ENDFOR

IF the current node is a pixel THEN
   IF all fields of snt are black THEN
       /* the current node is an internal pixel,
          ignore it */
       RETURN
   ELSE
       /* the current node is a border pixel */
       copy the current node into the output array
```

```
   ENDIF
ELSE IF the current node is a 2x2 node THEN
   CALL TwoByTwoBorder()
ELSE
   CALL NonTrivialBorder()
ENDIF
```

:end of BorderFind()

2.4.2 The routines for setting the *snt* bit vector

The section contains the routines that find neighbours and set the *snt* bit vector accordingly.

Setsnt():

```
/* Purpose: set snt according to neighbour types */

CALL Neighbour()
CALL NeighbourCheck()
set the fields of snt to black, white, or grey
```

:end of Setsnt()

Neighbour():

```
calculate the location code of the neighbour of equal size
```

:end of Neighbour()

The method of calculating the location code of the neighbour of equal size is that of *dilated integer arithmetic* [Schr 92], see Appendix A.

NeighbourCheck():

```
determine the type of the neighbour: black/white/grey
```

:end of NeighbourCheck()

The implementation of **NeighbourCheck()** is based on the binary search and the rules stated in [Yang 90], see Appendix A.

2.4.3 The routines for checking nodes larger than size 2×2

Any nodes larger than 2×2 require careful processing and are considered nontrivial.

NonTrivialBorder():

```
CALL SWBorder()
CALL SEBorder()
CALL NWBorder()
CALL NEBorder()
```

:end of NonTrivialBorder()

SWBorder():

```
/* Purpose: find the border pixels in the south-west
   quarter of the current node */
```

```
IF S1 and W1 fields of snt are both black THEN
   /* the edge pixels in the south-west quarter are
      all internal */
   RETURN
ELSE
   /* traverse the edge pixels of the south-west quarter */

   IF diff_border_LUT[g] is null THEN
      /* the first time a node of grouping factor g
         is encountered */
      initialize diff_border_LUT[g]
   ENDIF

   FOR each edge pixel in the SW quarter of current node DO
      IF OnBorder() THEN
         copy this pixel into the output array
      ENDIF
   ENDFOR
ENDIF
```

:end of SWBorder()

OnBorder():

```
/* Purpose: determine whether the current edge
   pixel 'cepix' is on the border */
```

```
IF snt for cepix indicates at least one white neighbour THEN
   RETURN TRUE
ELSE IF snt for cepix indicates all neighbours are black THEN
   RETURN FALSE
ELSE
   /* cepix is on the part of edge/corner that has
      grey neighbour(s) */
   CALL Neighbour()
   CALL NeighbourCheck()
   IF the particular neighbour of cepix is white THEN
      RETURN TRUE
   ELSE
      RETURN FALSE
   ENDIF
ENDIF
```

:end of OnBorder()

Note that the check of the S1 and W1 fields essentially replaces recursion. The lookup table diff_border_LUT[g] is used to generate the edge pixels of the south-west quarter. It is also used to determine on which side, S1, W1, or SW, the current edge pixel lies.

The other three routines **SEBorder()**, **NWBorder()**, and **NEBorder()** follow basically the same procedure as that of **SWBorder()**. The differences lie in which portions of the *snt* bit vector and the lookup table diff_border_LUT[g] are used.

2.4.4 The routine for checking 2×2 nodes

This section contains the routine that handles 2×2 nodes as a special case. It is treated as a special case because

1. examining the pixels of a 2×2 node is trivial and there is no need to use diff_border_LUT[1];
2. if all four pixels of the 2×2 node are on the border, they should be output as a single node instead of four individual pixels.

TwoByTwoBorder():

```
/* Purpose: find the border pixels of the current 2x2
   node using the information in snt */

IF all four pixels are on the border THEN
   copy the current node into the output array
ELSE
   FOR each pixel in the current node DO
      IF the pixel is on the border THEN
         copy the pixel into the output array
      ENDIF
   ENDFOR
ENDIF
```

:end of TwoByTwoBorder()

To determine whether a pixel is on the border, the information previously acquired in *snt* is used. For example, if the pixel examined is at the north-

east corner, then the N2 and E2 fields of *snt* are checked. If both of these two fields are black, that means the north and the east neighbours of the north-east corner pixel are black and hence the pixel is not on the border.

2.5 Further acceleration

The new algorithm is superior to the one by [Yang 90] in two aspects:

1. The border pixels of a PB-quadrant can be generated in sorted order without the need for an insertion sort.

2. The number of neighbour finding operations is reduced by avoiding recursion.

Since there is no recursion involved, the overhead associated with handling recursive calls is eliminated.

Unnecessary neighbour finding operations in the recursive method are avoided as well. For example, while checking the type of the east neighbour for the node P in Figure 2.3, the first neighbour finding operation determines that the east neighbour of equal size is grey. The recursive algorithm hence subdivides the node into its four children. The children involved in determining the east neighbour type are the south-east quarter and the north-east quarter. Each of these two child nodes will incur four more neighbour finding operations for the four directions. However, the west and the south neighbour checks for the north-east child node are unnecessary because its siblings are always black[5]. Similarly, the north and west neighbour checks for the south-east child node are not necessary.

[5]Recall that the current node considered is an input node and thus is black by definition.

The above improvements are obtained at the expense of auxiliary data structures and information keeping, e.g. the *snt* bit vector and the lookup tables diff_border_LUT.

Further acceleration is possible. Consider again the node P in Figure 2.3. When the north neighbour Q is checked and determined to be black, the N1 and N2 fields of the *snt* of P are set to black. But it is also clear that the south neighbour of Q is black. If the S1 and S2 fields of the *snt* of Q are set to black while examing P, there is no need to check the south neighbour of Q later on.

Chapter 3

Algorithm Correctness

3.1 Applicability of the algorithm

The new algorithm accepts as input a linear quadtree and outputs the border pixels as a linear quadtree. The border pixels produced are 8-connected.

An input image may consist of a single region or of multiply connected regions. Holes in an image are also allowed.

3.2 Verification of the algorithm

The routine **BorderFind()** examines each input node in turn, thus all possible border pixels will be visited. Holes in the image are therefore found as well in contrast to contour tracing algorithms such as the one of [Dyer 80] which cannot process holes and multiply-connected regions unless a seed is given for each border.

A border pixel must be an edge pixel (the converse is not necessarily true), hence, it is only necessary to examine the edge pixels of the input

nodes. An invariant is established to verify the correctness of the algorithm:

Invariant An edge pixel is on the border of the input quadtree if it has at least a white neighbour in one of its four directions: east, south, west, and north.

Each input node is associated with a *snt* bit vector, storing the types of the sub-neighbours in the four directions as shown in Figures 2.10 and 2.11. There are eight sections representing the eight sub-neighbours, two for each direction. For example, S1 faces the sub-neighbour which is the north-west child of the south neighbour; N2 faces the sub-neighbour which is the south-east child of the north neighbour. The values of *snt* provide the information for determining the neighbour types of the edge pixels. The correctness of the routines for setting *snt* is based on the rules described in [Yang 90] (see Appendix A) and the binary search algorithm.

After the *snt* fields are properly set, the input nodes are classified into three classes and are checked separately. They are:

- *pixel nodes*,
- 2×2 *nodes*, and
- *non-trivial nodes*, i.e. those that are larger than 2×2.

For pixel nodes, the only edge pixel is the node itself. Any two adjacent sections of the *snt* vector on the same side of the node contain the same value. For example, if the node in Figure 2.10 represents a pixel node, then the S1 and S2 sections contain the same value, either black or white. It is impossible for S1 and S2 to contain different *snt* values because a pixel cannot have a grey neighbour.

For the 2×2 class, the edge pixels are its four pixels; there are no non-edge pixels in the node. The width of each of the eight edge sections in a 2×2 node is that of a pixel, therefore E1 and E2, for example, can contain black and white respectively, meaning that the east neighbour is grey. However, no grey neighbour is possible for any of the eight sub-neighbours, i.e. E1 cannot contain grey value, neither can E2, since they are of pixel width.

The invariant is therefore not violated for the first two classes by checking the *snt* vector of each node. There will be no unresolved cases, i.e. grey sub-neighbours. The sub-neighbour types of black or white are sufficient to determine whether a pixel of a node is a border pixel. For example, if the W2 section of the *snt* vector of a 2×2 node contains the value 01, it means that the W2 sub-neighbour, which is the north-east child of the west neighbour, is white. Thus, the north-west pixel of the 2×2 node is a border pixel.

For nodes larger than 2×2, four subroutines will handle the edge pixels of the node. It is only necessary to consider the subroutine handling the south-west quarter for the correctness of the algorithm for the non-trivial class since the other three quarters (south-east, north-west, and north-east) have a structure similar to that of the south-west subroutine.

The south-west subroutine **SWBorder()** uses the values in the S1 and W1 sections of the *snt* vector. If both S1 and W1 contain the value 11, then all the edge pixels in the south-west quarter have black neighbours, no further processing is necessary. Otherwise, the individual edge pixels are traversed, using the lookup table diff_border_LUT[g], to determine whether they are adjacent to black, grey, or white neighbours. If the current edge pixel traversed is on the section S1, say, where its corresponding

sub-neighbour is

- black, then the current edge pixel is not a border pixel;
- white, then the current edge pixel is a border pixel;
- grey, then neighbour finding in the south direction is invoked for the current edge pixel to determine whether its south neighbour is white.

3.3 Experimental verification

Various linear-quadtree images were used as input test data; six of them and their borders are shown in Figures 3.1 to 3.12.

Careful examination of the figures shows the correctness of the algorithm.

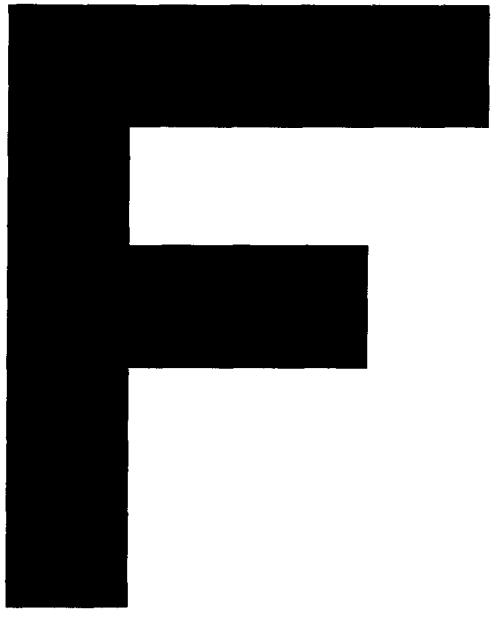

Figure 3.1: Quadtree **F**

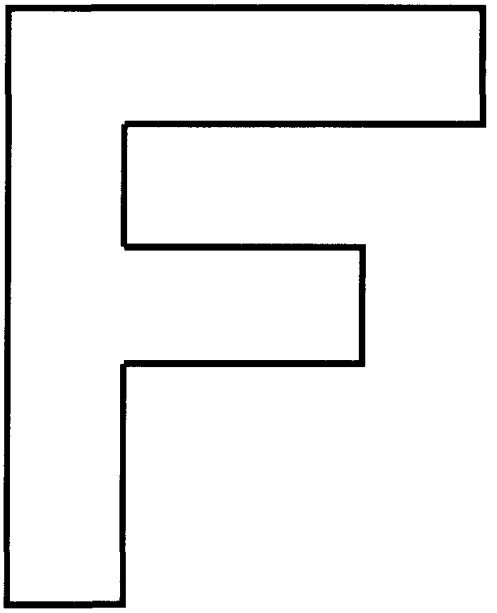

Figure 3.2: Borders of Quadtree **F**

Figure 3.3: Quadtree **slantF**

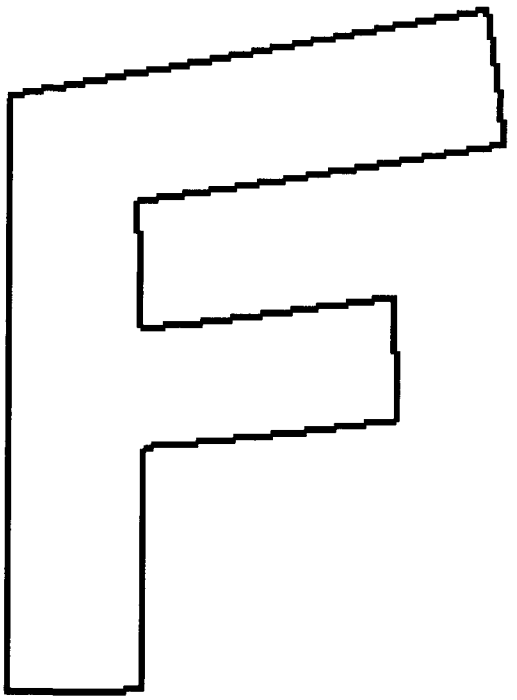

Figure 3.4: Borders of quadtree **slantF**

Figure 3.5: Quadtree **distortF**

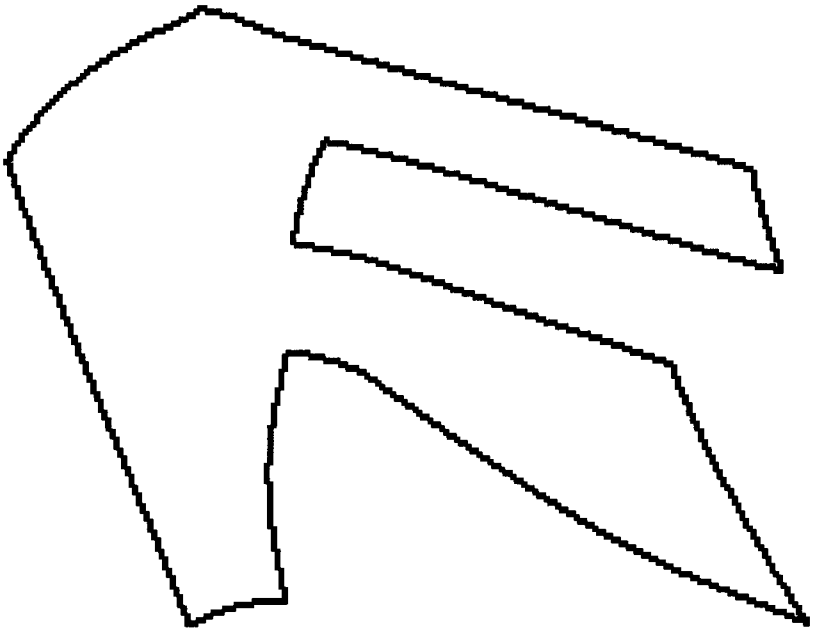

Figure 3.6: Borders of quadtree **distortF**

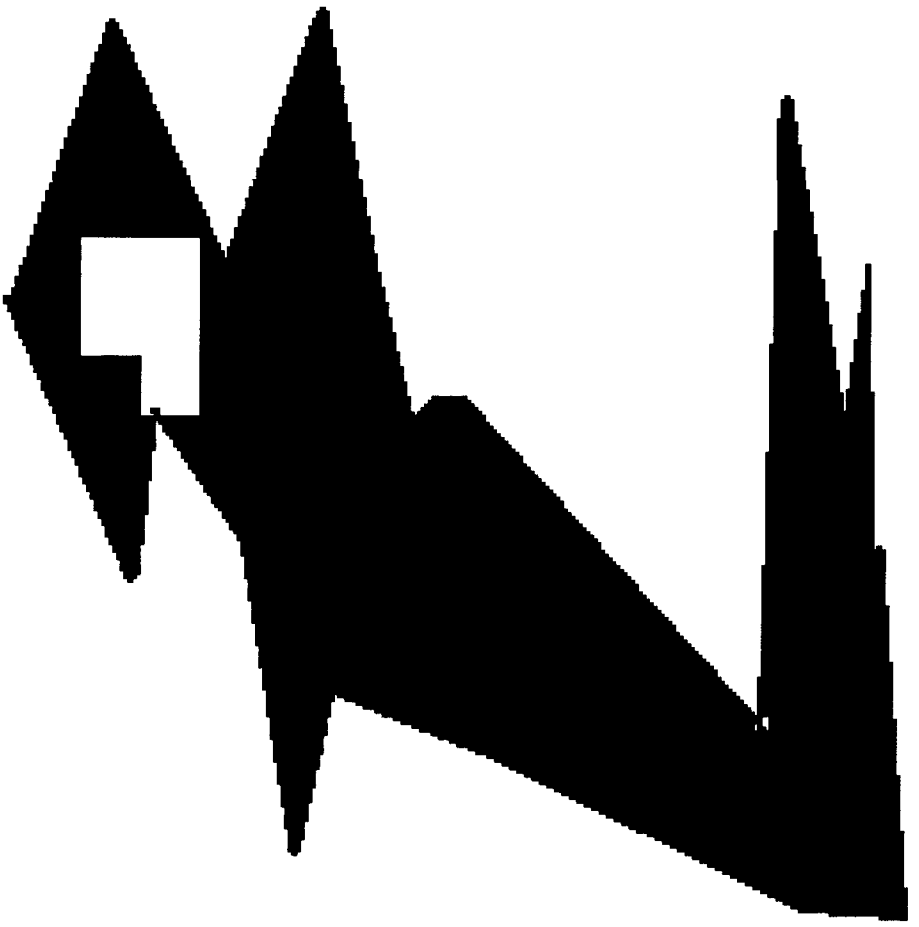

Figure 3.7: Quadtree **random1**

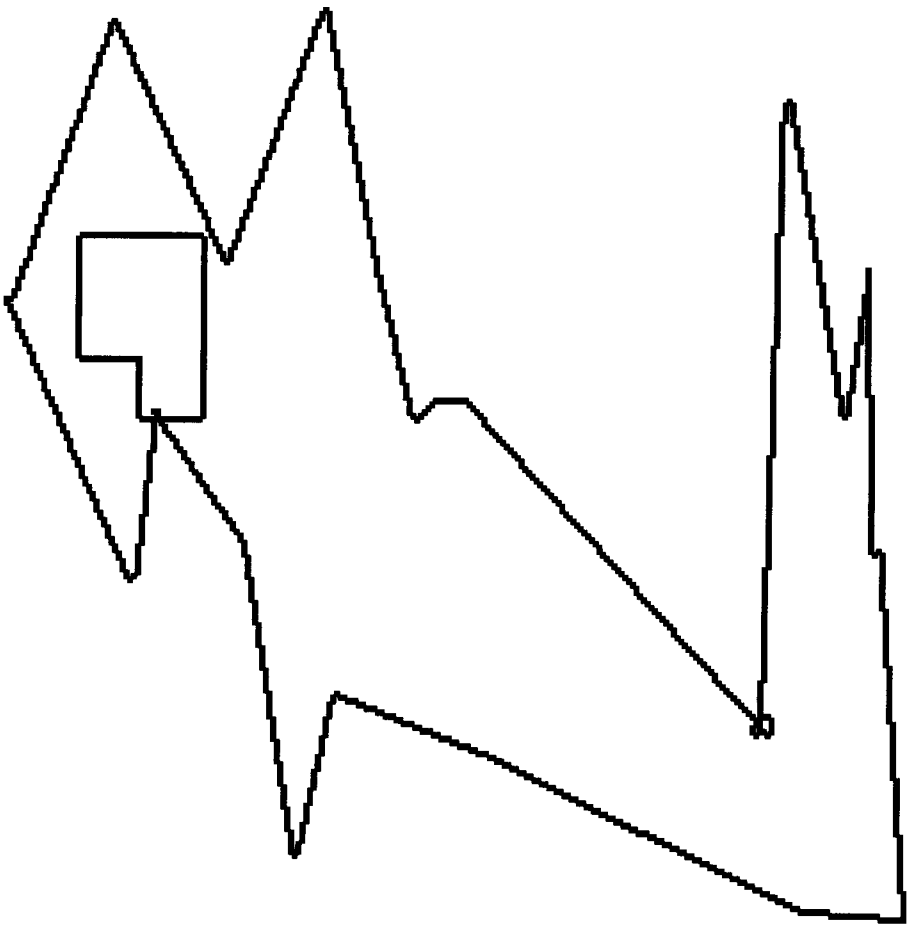

Figure 3.8: Borders of quadtree **random1**

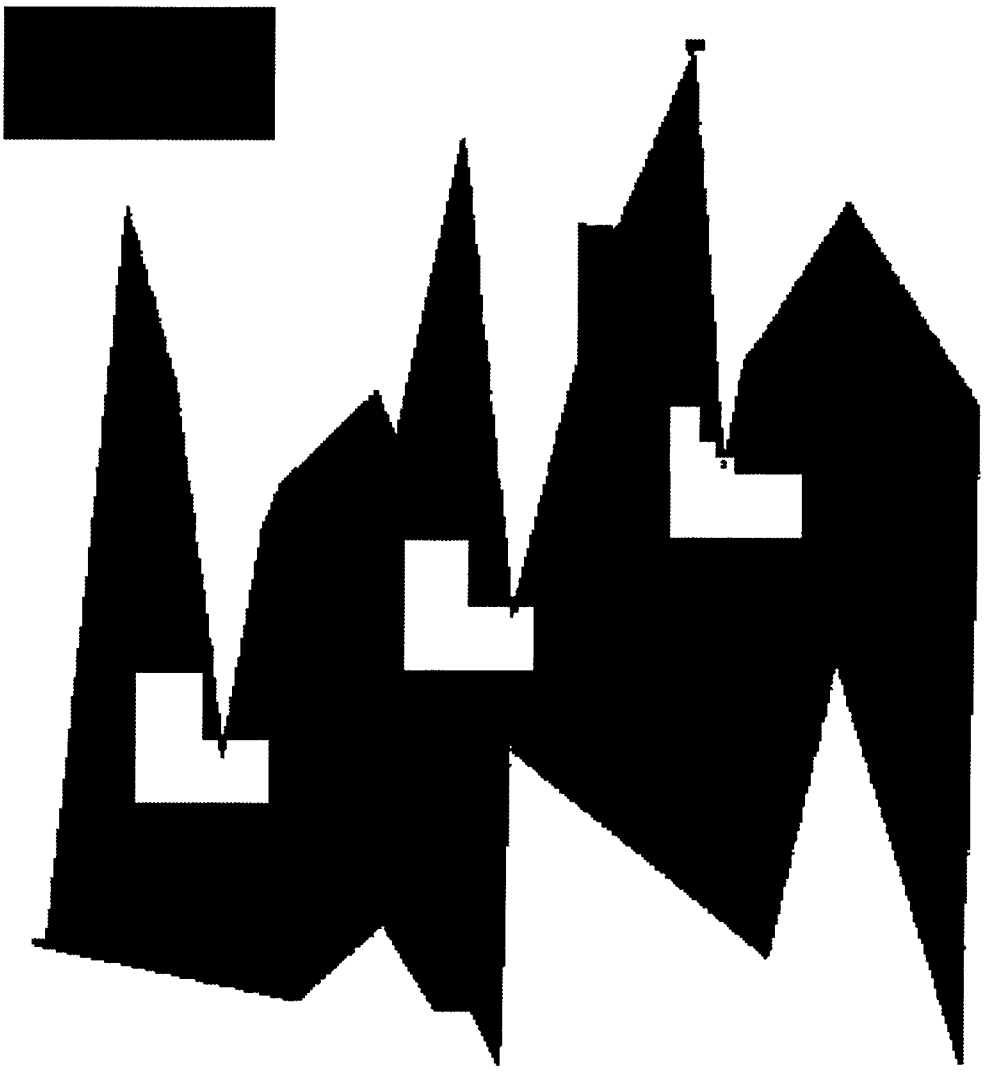

Figure 3.9: Quadtree **random2**

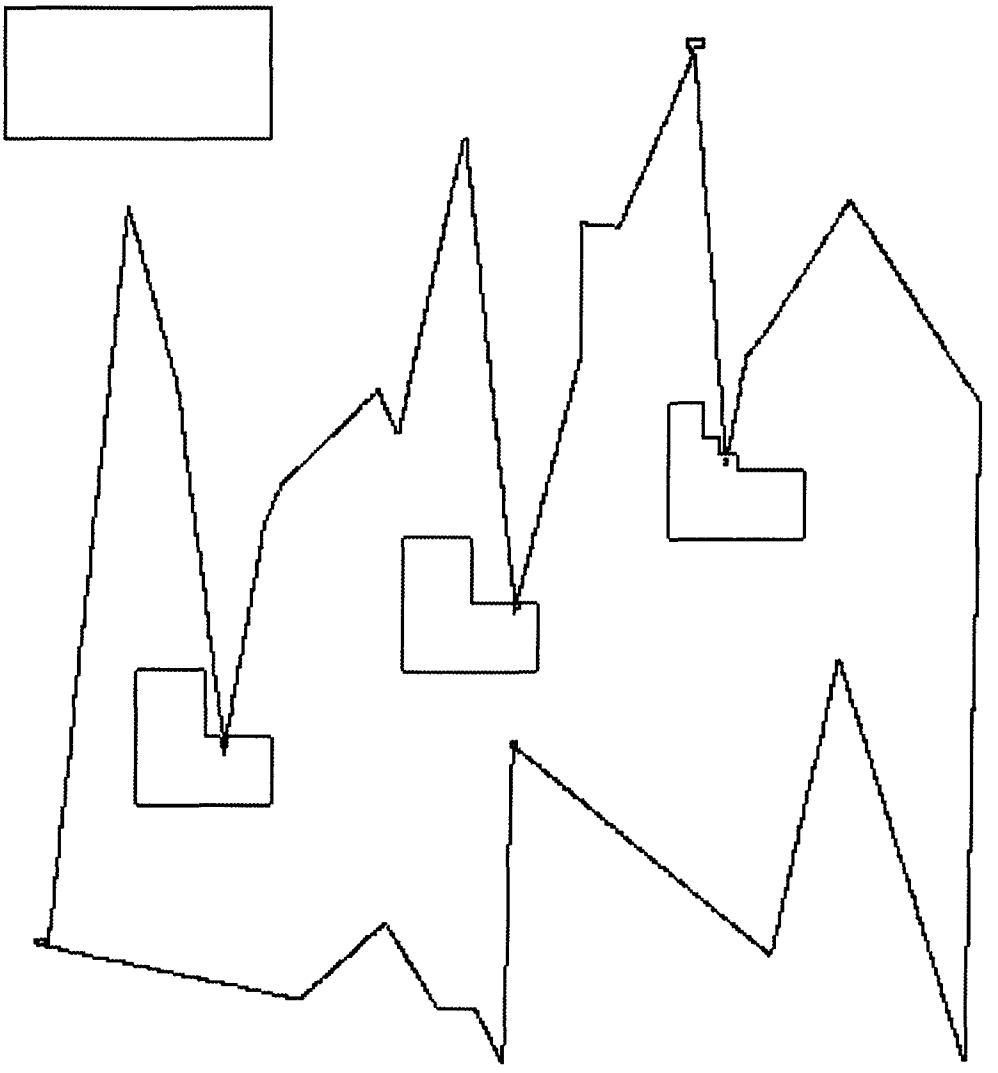

Figure 3.10: Borders of quadtree **random2**

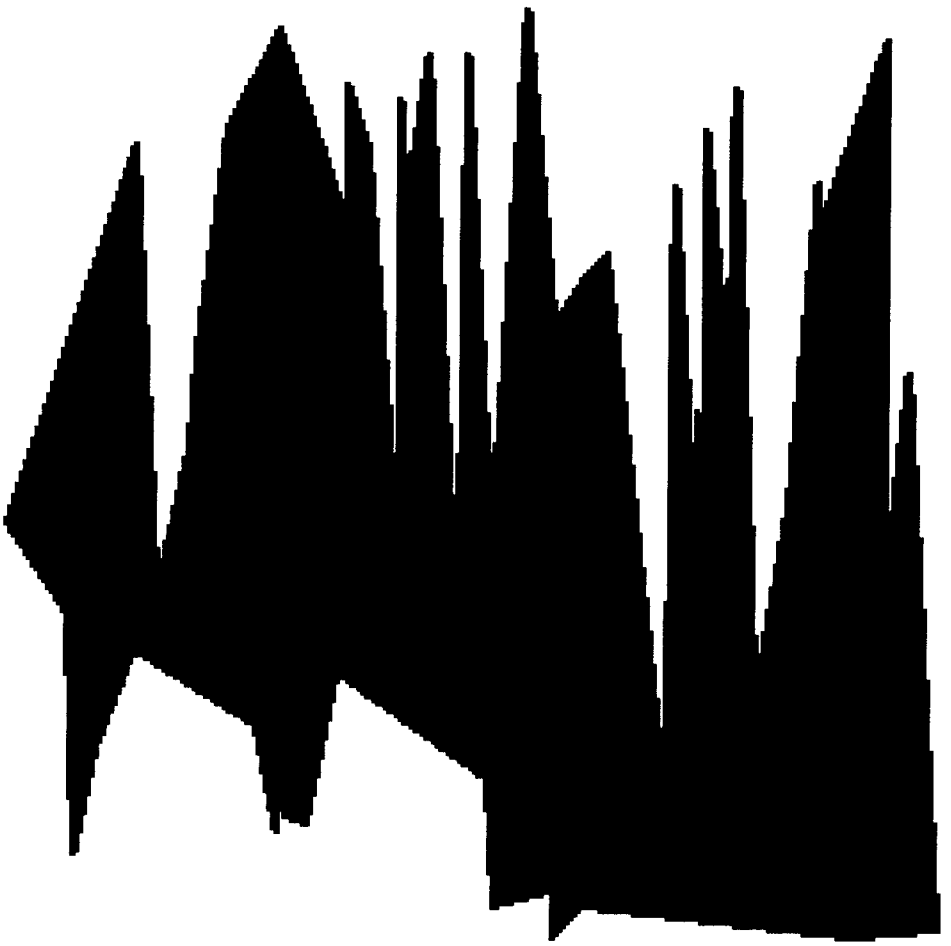

Figure 3.11: Quadtree **random3**

Figure 3.12: Borders of quadtree **random3**

Chapter 4

Algorithm Performance

4.1 Time complexity analysis

Based on the analysis of their algorithm in [Yang 90], Yang & Lin stated that the average number of neighbour finding operations for any given node is a constant. Each neighbour finding operation requires a binary search which is O(log N) where N is the number of nodes in the input quadtree. Thus, the algorithm would run in time O(N log N) on average. However, if predict-binary search, which is O(1) on average, is used instead of binary search, then their algorithm runs in O(N) on average.

The new algorithm also visits each of the N nodes. From the probabilistic analysis of Section 2.3, the new algorithm requires, on average, a smaller number of recursive subdivisions, and hence less neighbour finding operations, for a given node. Theoretically, the new algorithm also runs in O(N) on average if predict-binary search is used.

4.2 Runtime and profile

4.2.1 The implementation and run-time environment

The competing algorithm is that of Yang & Lin [Yang 90]. In order to compare the two algorithms, both were implemented in C and compiled using *gcc* with the optimization flag set. They were run on a **SPARC** *IPX* using the **Unix** system call **times()** for the timing results[1] and the command *gprof* for profiling the number of neighbour finding operations incurred.

There are two implementation details which are unclear in [Yang 90].

1. How is the predict-binary search implemented?

2. How are the border pixels of a PB-quadrant inserted, in sorted order, into the output linear quadtree?

Binary search, instead of predict-binary search, was used in both algorithms for the purpose of finding neighbours. Thus for comparison purposes, especially if the focus is in reducing the number of neighbour finding operations, the search method used is irrelevant.

A quick-sort routine was implemented for the second question because an insertion-sort routine would be too costly to use for an array implementation of the linear quadtree. However, which sorting method is used for the problem does not affect the number of neighbour finding operations incurred because outputting the border pixels of a PB-quadrant occurs after all neighbours have been found. Besides, the new algorithm does not require any kind of insertion or sorting, which is in principle an advantage.

[1] For a comprehensive illustration of the use of **times()**, see Section 8.16 of [Stev 92].

It is also noted that there is an error in the [Yang 90] algorithm. Consider the 2×2 node (60,1) with its black neighbours shown in Figure 4.1. The node is not considered a PB-quadrant and hence its four pixels are processed individually. The four pixels are all border pixels; however, they are not output as a single node, i.e. there is a need for condensation. Such an input node is handled correctly by the **TwoByTwoBorder()** routine in the new algorithm.

The new algorithm has been implemented in two versions, one with the additional acceleration of setting some fields of *snt* of the opposite black node, as described in Section 2.5. The other, **tang**, is without the additional acceleration. The first implementation is referred to as **atang**. The reason for the separate implementation is to isolate the non-recursive feature of the new algorithm in **tang** in order to determine clearly that the improvement, in terms of reducing the number of neighbour finding operations, stems mainly from the probabilistic analysis that leads to non-recursion.

The implementation of the algorithm in [Yang 90] will be called **yl**.

4.2.2 The results

Tables 4.1 to 4.3 show the performance of each program. Note that the symbol g represents the term grouping factor and NB is short for neighbour finding operations.

Tables 4.4 and 4.5 show the percentage improvements of the new algorithm over the one in [Yang 90].

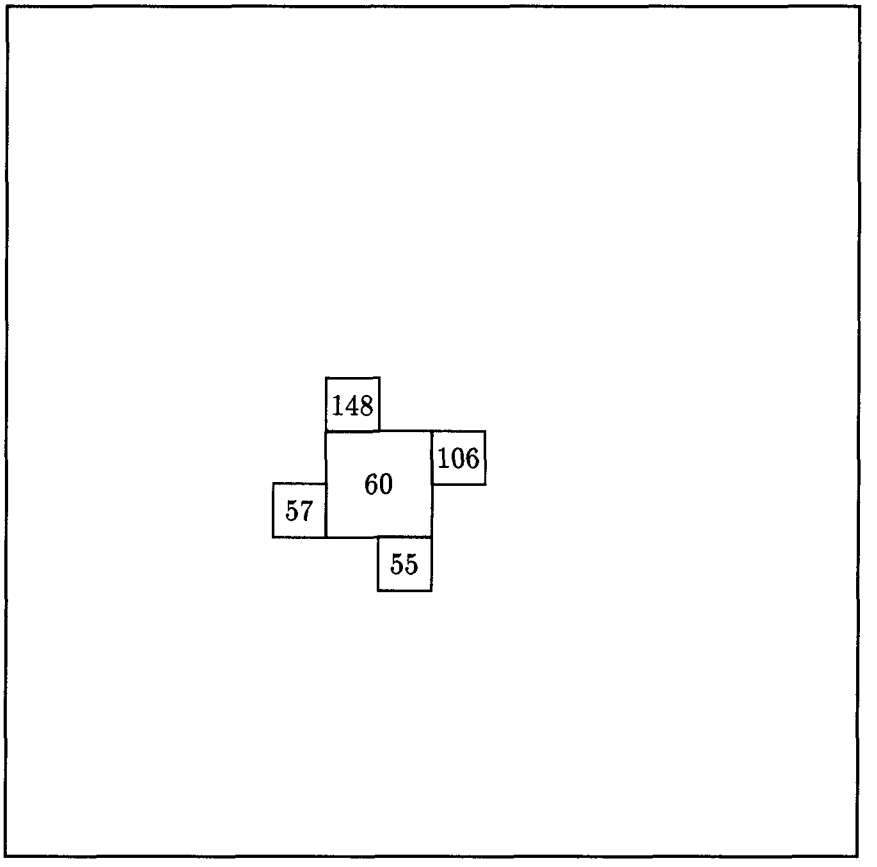

Figure 4.1: A 2×2 node where **yl** fails to condense

Table 4.1: Performance of the program **yl**

	F	**slantF**	**distortF**	**random1**	**random2**	**random3**
resolution	8	8	8	8	10	8
# of input nodes	580	633	1174	1545	7385	3097
# of border nodes	697	719	900	1472	8513	2766
max. g	4	4	5	5	7	5
user CPU (sec.)	0.15	0.17	0.32	0.40	2.33	0.78
sys CPU (sec.)	0.02	0.03	0.02	0.01	0.12	0.03
# of NB	8112	9236	16888	21204	108923	37892

Table 4.2: Performance of the program **tang**

	F	**slantF**	**distortF**	**random1**	**random2**	**random3**
resolution	8	8	8	8	10	8
# of input nodes	580	633	1174	1545	7385	3097
# of border nodes	697	719	900	1466	8507	2739
max. g	4	4	5	5	7	5
user CPU (sec.)	0.09	0.10	0.19	0.27	1.40	0.48
sys CPU (sec.)	0.01	0.02	0.01	0.01	0.05	0.03
# of NB	3115	3617	6904	8890	45615	16597

Table 4.3: Performance of the program **atang**

	F	**slantF**	**distortF**	**random1**	**random2**	**random3**
resolution	8	8	8	8	10	8
# of input nodes	580	633	1174	1545	7385	3097
# of border nodes	697	719	900	1466	8507	2739
max. g	4	4	5	5	7	5
user CPU (sec.)	0.08	0.09	0.17	0.22	1.25	0.42
sys CPU (sec.)	0.01	0.02	0.01	0.02	0.05	0.03
# of NB	2324	2729	5440	6923	36268	12353

Table 4.4: Percentage improvement of **tang** over **yl**

	F	**slantF**	**distortF**	**random1**	**random2**	**random3**
CPU (user & sys)	41%	40%	41%	32%	41%	37%
# of NB	62%	61%	59%	58%	58%	56%

Table 4.5: Percentage improvement of **atang** over **yl**

	F	**slantF**	**distortF**	**random1**	**random2**	**random3**
CPU (user & sys)	47%	45%	47%	41%	47%	44%
# of NB	71%	70%	68%	67%	67%	67%

4.2.3 Confidence interval

It would require more than 400 sample linear quadtrees for a 95% confidence interval on the performance comparison to reach a sampling error of $\pm 5\%$ [Kell 89], which is beyond the time frame for the current research. Therefore, a plot of the run-time or number of neighbour finding operations versus resolution is not appropriate.

From Tables 4.1 to 4.5, it is clear that the new algorithm has reached the goal which was set out to achieve: reducing the number of neighbour finding operations.

From Tables 4.4 and 4.5, it can be seen that the **tang** implementation achieves over 50% reduction in the number of neighbour finding operations, while the **atang** implementation achieves over 60%. The additional acceleration thus yields approximately a 10% advantage.

It is also clear from the tables that the overhead of auxiliary data structures does pay off as the run times of the new algorithm, in either implementation, is better than the one by Yang & Lin, viz. 30% to 40% improvement.

There are cases for which the new algorithm incurs more neighbour find-

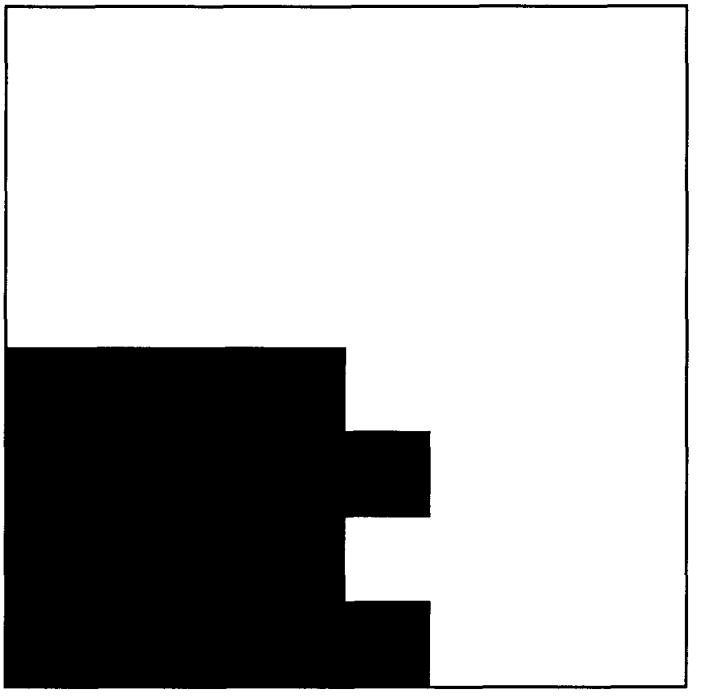

Figure 4.2: A case that **atang** performs poorer than **yl**

ing operations than that of Yang & Lin. Consider the quadtree with resolution 7 in Figure 4.2, the number of neighbour finding operations is 74 for both **atang** and **tang**, whereas **yl** only incurs 51. However, the same configuration but with resolution 6 produces the opposite result: 42 neighbour finding operations for **tang** and **atang** and 51 for **yl**.

A worst case for the new algorithm, as suggested by Figure 4.2, is when a node of grouping factor 6 or larger has neighbour nodes of grouping factor 2 less (not 1 less, not 3 less). The new algorithm prevails when an image has more pixels in general, which is also apparent from Equations 2.5 to 2.7 in that p_0 is larger with more pixels in the image. From the sample random

quadtrees, this favorable case seems to be common.

Chapter 5

Conclusion & Future Directions

A new border finding algorithm for linear quadtree based images is presented. It is able to generate the border pixels in sorted order and avoids sorting and recursion. By avoiding recursion, the new algorithm succeeds in reducing the number of neighbour finding operations when compared to the Yang & Lin algorithm which is based on recursion. The overhead incurred pays off as the run time of the new algorithm is also better in the sample test cases.

Future directions for the research could include

1. More sample quadtrees shall be used for testing, which would require a more robust and versatile random border generator.

2. Implement an algorithm with $m = 0$ as discussed in Section 2.3. With $m = 0$, there is no need for the *snt* bit vector. Thus both the program space and the heap space can be reduced; however, whether it will be

faster is still unknown.

3. Extend the new algorithm to linear octree based 3D images. The idea of *snt* and thus non-recursion is directly applicable to linear octrees. However, generating the edge pixels in sorted order would require a 3D model of a node with grouping factor 4 to help visualize the edge pixels.

4. Extend the new algorithm to grey scale or colour images.

5. Consider the possibility of converting the algorithm for parallel processing.

Chapter 6

Glossary

E1 The first section of east, see Figure 2.10

E2 The second section of east, see Figure 2.10

g Grouping factor

LUT Lookup table

loc Location code

N1 The first section of north, see Figure 2.10

N2 The second section of north, see Figure 2.10

NE North-east

NW North-west

PB Proper border

p_i Probability of finding a neighbour with grouping factor i.

S1 The first section of south, see Figure 2.10

S2 The second section of south, see Figure 2.10

SE South-east

SW South-west

snt Sub-neighbour type, see Figure 2.11

W1 The first section of west, see Figure 2.10

W2 The second section of west, see Figure 2.10

Bibliography

[Atki 84] Atkinson, H.H., I. Gargantini, M.V.S. Ramanath. **Determination of the 3D border by repeated elimination of internal surfaces**, *Computing*, 32, 4, pp. 279–295 (October 1984).

[Atki 85] Atkinson, H.H., I. Gargantini, M.V.S. Ramanath. **Improvements to a recent 3D-border algorithm**, *Pattern Recognition*, 18, 3-4, pp. 215–226 (March 1985).

[Dill 88] Dillencourt, M.B., H. Samet. **Extracting region boundaries from maps stored as linear quadtrees**, *Proceedings Third International Symposium on Spatial Data Handling*, Sydney, Australia, pp. 65–77 (August 1988).

[Dyer 80] Dyer, C.R., A. Rosenfeld, H. Samet. **Region representation: boundary codes from quadtrees**, *Communications of ACM*, 23, 3, pp. 171–179 (March 1980).

[Fran 91] Franciosa, P.G., E. Nardelli. **A guaranteed approximation algorithm for on-line computing quadtree border**, *Geographic Database Management Systems, Workshop Proceedings*, Capri, Italy, pp. 245–257 (May 1991).

[Garg 82] Gargantini, I. **An effective way to represent quadtrees,** *Communications of ACM*, 25, pp. 905–910 (1982).

[Kell 89] Kelly, B., B. Alexander, P. Atkinson, J. Swift. *Mathematics 11*, British Columbia Edition, Addison-Wesley (1989).

[Li 84] Li, S.-H., M.H. Loew. **Boundary chain codes from quadcode-trees,** *Proceedings IEEE Workshop on Computer Vision, Representation and Control*, Annapolis, MA, pp. 178–182 (April 1984).

[Pavl 82] Pavlidis, T. *Algorithms for Graphics and Image Processing*, Computer Press (1982).

[Qian 89] Qian, K., P. Bhattacharya. **A polynomial approach to image processing and quadtrees,** *IEEE Eighth Annual International Phoenix Conference Proceedings of Computers and Communications*, pp. 596–600 (1989).

[Rose 82] Rosenfeld, A., A.C. Kak. *Digital Picture Processing*, 2nd ed., Academic Press (1982).

[Same 90] Samet, H. *Applications of Spatial Data Structures: Computer Graphics, Image Processing and GIS*, Addison-Wesley, Reading, MA (1990).

[Schr 92] Schrack, G.F. **Finding neighbours of equal size in linear quadtrees and octrees in constant time,** *CVGIP: Image Understanding*, 55, 3, pp. 221–230 (May 1992).

[Stev 92] Stevens, W.R. *Advanced Programming in the UNIX Environment*, Addison-Wesley (1992).

[Yang 90] Yang, S.-N., T.-W. Lin. **A new 3D border algorithm by neighbour finding,** *Proceedings Fourteenth Annual International Computer Software and Applications Conference*, Chicago, IL, pp. 353–358 (October 1990).

Appendix A

Program Listing

The complete program listing for the new border finding algorithm is included in the appendix. However, the main ideas used to implement the routines **NeighbourCheck()** and **Neighbour()** are explained first.

Let LQT be the set of all linear quadtree nodes. Define the *d-edge* of a quadrant Q to be the unique edge of Q in the direction $d \in \{N, S, E, W\}$. The quadrant Q_d is the *d-neighbour* of quadrant Q if it has the same size as Q and the d-edge of Q is shared by the $\bar{d}$-edge of Q_d, where $\bar{d}$ is the opposite direction of d in usual sense. If Q_d is white, then Q has a *white neighbour* Q_d and therefore its d-edge is a border edge. If Q_d is grey, then Q_d contains at least one proper sub-quadrant Q_* in the linear quadtree. Therefore in this case, Q has a *grey neighbour* Q_d and Q has a *sub-neighbour* Q_*. If Q_d is black then either Q_d is a linear quadtree node or Q_d is properly contained in a super-quadrant Q^* of the linear quadtree. For the former case Q_d is called an *exact neighbour* of quadrant Q and the latter case Q^* is called a *super-neighbour* of quadrant Q. The *least upper bound* of a quadrant Q is

defined to be $lub(Q) = min\{B \in LQT : Q \leq B\}$. The *greatest lower bound* of Q is defined to be $glb(Q) = max\{B \in LQT : B \leq Q\}$.

Yang & Lin made a few observations in [Yang 90] that help determine the type of a neighbour and facilitate the implementation of the routine **NeighbourCheck()**.

1. Let $Q \in LQT$ and its d-neighbour be Q_d. Then $glb(Q_d)$ is a super-neighbour if and only if $glb(Q_d) \neq Q_d$ and $glb(Q_d) \cap Q_d \neq \phi$.

2. Let $Q \in LQT$ and its d-neighbour be Q_d. Then $lub(Q_d)$ is a sub-neighbour if and only if Q_d is a grey node.

3. Let the grouping factors of quadrants Q_1 and Q_2 be g_1 and g_2 respectively. Then $Q_1 \subset Q_2$ if

   (a) $g_1 \leq g_2$, and

   (b) $loc_1 \oplus loc_2 < 4^{g_2}$, where loc_1 and loc_2 are the location codes of Q_1 and Q_2 respectively, and $\oplus$ is the exclusive OR operator.

Schrack formulated a few theorems in [Schr 92] that allow efficient calculation of neighbour codes of equal size and thus facilitate the implementation of the routine **Neighbour()**.

Theorem 1 *Let* quad location addition $\oplus_q$ *be the separate addition of the x- and y-components of a linear quadtree node* n_q *and of a translation increment (relative coordinates)* Δn_q*, and the recombination of the results into a location code. Then*

$$n_q \oplus_q \Delta n_q = (((n_q \mid t_y) + (\Delta n_q \wedge t_x)) \wedge t_x) \mid (((n_q \mid t_x) + (\Delta n_q \wedge t_y)) \wedge t_y),$$

where $t_x = \sum_{i=0}^{r} 4^i$, $t_y = t_x \ll 1$, *and* $\ll$, $|$, $\wedge$ *are the machine left shift, OR, AND operators respectively.*

Theorem 2 *Given a location code* n_q *and its level* l, *the eight neighbours of equal size (level* l*) are given by*

$$m_q = n_q \oplus_q (\Delta n_i \ll (2(r-1))), i = 0, \ldots, 7,$$

where $\oplus_q$ *is the quad location addition operator,* Δn_i *are the eight basic direction increments that are encoded as location codes,* $\ll$ *is the machine left shift operator, and* r *is the (fixed) resolution.*

```
/********************************** qutil.h ***************************
 * Header file for quadtree-related problems
 * Date         By              Comment
 * ----         --              -------
 * 1994Feb11    S.K.Tang        Created
 */

/* dimension is 2D for quadtree */
#define DIM      2

/* default number of nodes for input and output quadtrees */
#define DEFAULT_NUM_IN_NODES    2000
#define DEFAULT_NUM_OUT_NODES   1000

/* location code type */
typedef long int        LocCode;

/* boolean type */
typedef char            Boolean;
#define TRUE    1
#define FALSE   0

/* node of a quadtree */
typedef unsigned int    snt_type;
typedef struct {
        LocCode         loc;    /* location code */
        int             lev;    /* level: lev=resolution for pixel,
                                          lev=0 for root */
        int             gf;     /* grouping factor: gf=0 for pixel,
                                   gf=resolution for root */
        snt_type        snt;    /* bit vector for sub-neighbour type*/
} QtNode;

/* values for snt_type (see 4/1/94 research note) */
#define E1_WHITE        0x0001
#define E1_GRAY         0x0002
#define E1_BLACK        0x0003
#define E2_WHITE        0x0004
#define E2_GRAY         0x0008
#define E2_BLACK        0x000c
#define S1_WHITE        0x0010
#define S1_GRAY         0x0020
#define S1_BLACK        0x0030
#define S2_WHITE        0x0040
#define S2_GRAY         0x0080
#define S2_BLACK        0x00c0
#define W1_WHITE        0x0100
#define W1_GRAY         0x0200
#define W1_BLACK        0x0300
#define W2_WHITE        0x0400
#define W2_GRAY         0x0800
#define W2_BLACK        0x0c00
#define N1_WHITE        0x1000
#define N1_GRAY         0x2000
#define N1_BLACK        0x3000
#define N2_WHITE        0x4000
#define N2_GRAY         0x8000
#define N2_BLACK        0xc000
#define UNSET           0x0000

                        /* does the m portion of snt t
                                matches color c */
#define MATCH_COLOR(t,m,c)      (((t) & (m)) == (c))

                        /* is snt t all WHITE */
#define ALL_WHITE(t)            ((t) == 0x5555)

                        /* is snt t all BLACK */
#define ALL_BLACK(t)            ((t) == 0xffff)

/* a quadtree */
typedef struct {
        QtNode          *node;          /* nodes of the quadtree,
```



```
                                        should be pointer
                                        to array of QtNode */
        long int        size;           /* # of QtNode in quadtree */
        long int        allocated;      /* # of QtNode allocated */
} Qt;

/* integral Cartesian point */
typedef struct {
        long int        v[DIM];
} Point;

/* neighbour directions */
typedef int     DirType;
#define EAST            0
#define NORTH_EAST      1
#define NORTH           2
#define NORTH_WEST      3
#define WEST            4
#define SOUTH_WEST      5
#define SOUTH           6
#define SOUTH_EAST      7

/* neighbour types */
typedef int     NeiType;
#define BLACK_EXACT     0       /* black and exact neighbour */
#define BLACK_NOTEXACT  1       /* black and not exact neighbour */
#define WHITE           2       /* white neighbour */
#define GRAY            3       /* gray neighbour */

                        /* is the neighbour type nt black */
#define IS_BLACK(nt)  (((nt)==BLACK_EXACT) || ((nt)==BLACK_NOTEXACT))

/* sub-border sections (see 4/1/94 research note) */
typedef int     SubBorderType;
#define E1W2    0
#define SW      1
#define S1      2
#define W1      3
#define S2      4
#define SE      5
#define E1      6
#define W2      7
#define NW      8
#define N1      9
#define E2      10
#define N2      11
#define NE      12
#define SIZE_OF_SubBorderType   13

/* an edge pixel of a node */
typedef struct {
        LocCode         loc;    /* loc. code of this edge pixel */
        SubBorderType   sb;     /* sub-border direction of this
                                        pixel */
} BorPix;


/***************************** newborder.c ****************************
 * A new border finding algorithm by Tang and Schrack
 * Date         by              Comment
 * ----         --              -------
 * 1994Apr5     tang            Created
 * 1994May26    tang            Use PrTimes() for timing
 */
#include <stdio.h>
#include <sys/times.h>
#include "../header/qutil.h"

/* externals */
extern  long int        res;    /* resolution */
extern  LocCode         *pow4;  /* pow4[i] = 4^(res-i) */
                        /* differences of loc. codes of node edge */
extern  BorPix          **diff_border_LUT;
```

```
extern  QtNode          Neighbour();
extern  NeiType         NeighbourCheck();
extern  void            PrTimes();

void Init( inp, outp, in_fp, out_fp )
        Qt      *inp;   /* input quadtree */
        Qt      *outp;  /* output quadtree */
        FILE    *in_fp; /* input file pointer */
        FILE    *out_fp;/* output file pointer */
/*-------------------------------------------------------------
 * Reading input quadtree and initialize various data structures
 */
{
        GetInputQt( inp, in_fp );
        AllocOutputQt( outp, out_fp );
        InitPow4();
        InitMask();
        InitCoding();
        InitNeighbourLUT();
        InitBorder();
}

void SetNeighboursntBLACK( dir, npos, inp )
        DirType         dir;    /* neighbour direction */
        long int        npos;   /* current neighbour position */
        Qt              *inp;   /* input quadtree */
/*-------------------------------------------------------------
 * Set the sub-neighbour type opposite to the direction dir to
 * be BLACK for the neighbour node inp->node[npos]
 */
{
        switch( dir ) {
                case EAST:
                        if ( MATCH_COLOR(inp->node[npos].snt,
                                                W1_BLACK,UNSET) ) {
                                inp->node[npos].snt |= W1_BLACK;
                                inp->node[npos].snt |= W2_BLACK;
                        }
                        break;
                case SOUTH:
                        if ( MATCH_COLOR(inp->node[npos].snt,
                                                N1_BLACK,UNSET) ) {
                                inp->node[npos].snt |= N1_BLACK;
                                inp->node[npos].snt |= N2_BLACK;
                        }
                        break;
                case WEST:
                        if ( MATCH_COLOR(inp->node[npos].snt,
                                                E1_BLACK,UNSET) ) {
                                inp->node[npos].snt |= E1_BLACK;
                                inp->node[npos].snt |= E2_BLACK;
                        }
                        break;
                case NORTH:
                        if ( MATCH_COLOR(inp->node[npos].snt,
                                                S1_BLACK,UNSET) ) {
                                inp->node[npos].snt |= S1_BLACK;
                                inp->node[npos].snt |= S2_BLACK;
                        }
                        break;
        }
}

void SetsntBLACK( dir, cip, inp, sub )
        DirType         dir;    /* neighbour direction */
        long int        cip;    /* current input position */
        Qt              *inp;   /* input quadtree */
        unsigned int    sub;    /* sub-segment 1 or 2
                                        (other # for both) */
/*-------------------------------------------------------------
 * Set sub-neighbour type in direction dir to be BLACK
 * for the current node inp->node[cip]
 */
```



```
{
        switch( dir ) {
                case EAST:
                        if ( sub == 1 ) {
                                inp->node[cip].snt |= E1_BLACK;
                        } else if ( sub == 2 ) {
                                inp->node[cip].snt |= E2_BLACK;
                        } else {
                                inp->node[cip].snt |= E1_BLACK;
                                inp->node[cip].snt |= E2_BLACK;
                        }
                        break;
                case SOUTH:
                        if ( sub == 1 ) {
                                inp->node[cip].snt |= S1_BLACK;
                        } else if ( sub == 2 ) {
                                inp->node[cip].snt |= S2_BLACK;
                        } else {
                                inp->node[cip].snt |= S1_BLACK;
                                inp->node[cip].snt |= S2_BLACK;
                        }
                        break;
                case WEST:
                        if ( sub == 1 ) {
                                inp->node[cip].snt |= W1_BLACK;
                        } else if ( sub == 2 ) {
                                inp->node[cip].snt |= W2_BLACK;
                        } else {
                                inp->node[cip].snt |= W1_BLACK;
                                inp->node[cip].snt |= W2_BLACK;
                        }
                        break;
                case NORTH:
                        if ( sub == 1 ) {
                                inp->node[cip].snt |= N1_BLACK;
                        } else if ( sub == 2 ) {
                                inp->node[cip].snt |= N2_BLACK;
                        } else {
                                inp->node[cip].snt |= N1_BLACK;
                                inp->node[cip].snt |= N2_BLACK;
                        }
                        break;
        }
}

void SetsntGRAY( dir, cip, inp, sub )
        DirType         dir;    /* neighbour direction */
        long int        cip;    /* current input position */
        Qt              *inp;   /* input quadtree */
        unsigned int    sub;    /* sub-segment 1 or 2
                                        (other # for both) */
/*-------------------------------------------------------------
 * Set sub-neighbour type in direction dir to be GRAY
 * for the current node inp->node[cip]
 */
{
        switch( dir ) {
                case EAST:
                        if ( sub == 1 ) {
                                inp->node[cip].snt |= E1_GRAY;
                        } else if ( sub == 2 ) {
                                inp->node[cip].snt |= E2_GRAY;
                        } else {
                                inp->node[cip].snt |= E1_GRAY;
                                inp->node[cip].snt |= E2_GRAY;
                        }
                        break;
                case SOUTH:
                        if ( sub == 1 ) {
                                inp->node[cip].snt |= S1_GRAY;
                        } else if ( sub == 2 ) {
                                inp->node[cip].snt |= S2_GRAY;
                        } else {
```

```
                        inp->node[cip].snt |= S1_GRAY;
                        inp->node[cip].snt |= S2_GRAY;
                }
                break;
        case WEST:
                if ( sub == 1 ) {
                        inp->node[cip].snt |= W1_GRAY;
                } else if ( sub == 2 ) {
                        inp->node[cip].snt |= W2_GRAY;
                } else {
                        inp->node[cip].snt |= W1_GRAY;
                        inp->node[cip].snt |= W2_GRAY;
                }
                break;
        case NORTH:
                if ( sub == 1 ) {
                        inp->node[cip].snt |= N1_GRAY;
                } else if ( sub == 2 ) {
                        inp->node[cip].snt |= N2_GRAY;
                } else {
                        inp->node[cip].snt |= N1_GRAY;
                        inp->node[cip].snt |= N2_GRAY;
                }
                break;
        }
}

void SetsntWHITE( dir, cip, inp, sub )
        DirType         dir;    /* neighbour direction */
        long int        cip;    /* current input position */
        Qt              *inp;   /* input quadtree */
        unsigned int    sub;    /* sub-segment 1 or 2
                                   (other # for both) */
/*-------------------------------------------------------------
 * Set sub-neighbour type in direction dir to be WHITE
 * for the current node inp->node[cip]
 */
{
        switch( dir ) {
        case EAST:
                if ( sub == 1 ) {
                        inp->node[cip].snt |= E1_WHITE;
                } else if ( sub == 2 ) {
                        inp->node[cip].snt |= E2_WHITE;
                } else {
                        inp->node[cip].snt |= E1_WHITE;
                        inp->node[cip].snt |= E2_WHITE;
                }
                break;
        case SOUTH:
                if ( sub == 1 ) {
                        inp->node[cip].snt |= S1_WHITE;
                } else if ( sub == 2 ) {
                        inp->node[cip].snt |= S2_WHITE;
                } else {
                        inp->node[cip].snt |= S1_WHITE;
                        inp->node[cip].snt |= S2_WHITE;
                }
                break;
        case WEST:
                if ( sub == 1 ) {
                        inp->node[cip].snt |= W1_WHITE;
                } else if ( sub == 2 ) {
                        inp->node[cip].snt |= W2_WHITE;
                } else {
                        inp->node[cip].snt |= W1_WHITE;
                        inp->node[cip].snt |= W2_WHITE;
                }
                break;
        case NORTH:
                if ( sub == 1 ) {
                        inp->node[cip].snt |= N1_WHITE;
                } else if ( sub == 2 ) {
```



```
                        inp->node[cip].snt |= N2_WHITE;
                } else {
                        inp->node[cip].snt |= N1_WHITE;
                        inp->node[cip].snt |= N2_WHITE;
                }
                break;
        }
}

void SetChildsnt( dir, cip, inp, q, sub )
        DirType         dir;    /* neighbour direction */
        long int        cip;    /* current input position */
        Qt              *inp;   /* input quadtree */
        QtNode          q;      /* a child node */
        unsigned int    sub;    /* sub-segment 1 or 2
                                   (other # for both) */
/*-------------------------------------------------------------
 * Set the sub-neighbour type for child q in the direction dir
 * (which correspond to sub-segment sub in direction dir)
 */
{
        QtNode          n;      /* neighbour node */
        NeiType         nt;     /* neighbour type */
        long int        npos;   /* index position of neighbour
                                   if in inp */

        n = Neighbour( q, dir );
        nt = NeighbourCheck( q, n, dir, inp, &npos );
        switch( nt ) {
                case WHITE:
                        SetsntWHITE( dir, cip, inp, sub );
                        break;
                case GRAY:
                        SetsntGRAY( dir, cip, inp, sub );
                        break;
                case BLACK_EXACT:
                        SetNeighboursntBLACK( dir, npos, inp );
                case BLACK_NOTEXACT:
                        SetsntBLACK( dir, cip, inp, sub );
                        break;
        }
}

void CheckSubNeighbours( dir, cip, inp )
        DirType         dir;    /* neighbour direction */
        long int        cip;    /* current input position */
        Qt              *inp;   /* input quadtree */
/*-------------------------------------------------------------
 * Check the sub-neighbours of current node inp->node[cip] to
 * set their sub-neighbour types
 */
{
        LocCode         loc;    /* location code of current node */
        QtNode          q;      /* a child node */

        loc = inp->node[cip].loc;
        q.lev = inp->node[cip].lev + 1;
        q.gf = inp->node[cip].gf - 1;
        switch( dir ) {
                case EAST:
                        q.loc = loc + pow4[q.lev];
                        SetChildsnt( dir, cip, inp, q, 1 );
                        q.loc = loc + 3*pow4[q.lev];
                        SetChildsnt( dir, cip, inp, q, 2 );
                        break;
                case SOUTH:
                        q.loc = loc;
                        SetChildsnt( dir, cip, inp, q, 1 );
                        q.loc = loc + pow4[q.lev];
                        SetChildsnt( dir, cip, inp, q, 2 );
                        break;
                case WEST:
                        q.loc = loc;
```

```
                        SetChildsnt( dir, cip, inp, q, 1 );
                        q.loc = loc + 2*pow4[q.lev];
                        SetChildsnt( dir, cip, inp, q, 2 );
                        break;
                case NORTH:
                        q.loc = loc + 2*pow4[q.lev];
                        SetChildsnt( dir, cip, inp, q, 1 );
                        q.loc = loc + 3*pow4[q.lev];
                        SetChildsnt( dir, cip, inp, q, 2 );
                        break;
        }
}

void Setsnt( dir, cip, inp )
        DirType         dir;    /* neighbour direction */
        long int        cip;    /* current input position */
        Qt              *inp;   /* input quadtree */
/*-------------------------------------------------------------
 * Set sub-neighbour type in direction dir for the current node
 * inp->node[cip]
 */
{
        QtNode          n;      /* neighbour node */
        NeiType         nt;     /* neighbour type */
        long int        npos;   /* index position of neighbour
                                   if in inp */

        n = Neighbour( inp->node[cip], dir );
        nt = NeighbourCheck( inp->node[cip], n, dir, inp, &npos );
        switch ( nt ) {
                case WHITE:
                        SetsntWHITE( dir, cip, inp, 3 );
                        break;
                case GRAY:
                        CheckSubNeighbours( dir, cip, inp );
                        break;
                case BLACK_EXACT:
                        SetNeighboursntBLACK( dir, npos, inp );
                case BLACK_NOTEXACT:
                        SetsntBLACK( dir, cip, inp, 3 );
                        break;
        }
}

Boolean OnABorder( bpix, cip, inp, dir, white, black )
        BorPix          bpix;   /* an edge pixel of a node */
        long int        cip;    /* current input position */
        Qt              *inp;   /* input quadtree */
        DirType         dir;    /* direction */
        snt_type        white;  /* white snt */
        snt_type        black;  /* black snt */
/*-------------------------------------------------------------
 * Determine whether bpix is on the border of quadtree inp.
 * bpix is on a side or vertex of node edge of inp->node[cip]
 * in the direction dir with white or black (or gray) snt
 */
{
        QtNode          bqpix;  /* the same as bpix,
                                   but put as QtNode */
        QtNode          n;      /* neighbour node */
        NeiType         nt;     /* neighbour type */
        long int        npos;   /* index position of neighbour
                                   if in inp */

        bqpix.loc = bpix.loc;
        bqpix.lev = res;
        bqpix.gf = 0;
        if ( MATCH_COLOR(inp->node[cip].snt,black,white) ) {
                return TRUE;
        } else if ( MATCH_COLOR(inp->node[cip].snt,black,black) ) {
                return FALSE;
        } else {
                /* it is on gray, determine the neighbour
```



```
                   type of bpix only */
                n = Neighbour( bqpix, dir );
                nt = NeighbourCheck( bqpix, n, dir, inp, &npos );
                switch( nt ) {
                        case WHITE:
                                return TRUE;
                        case GRAY:
                                fprintf( stderr, "OnABorder()\n" );
                                exit( -4 );
                        case BLACK_EXACT:
                                SetNeighboursntBLACK( dir, npos, inp);
                        case BLACK_NOTEXACT:
                                return FALSE;
                }
        }
}

Boolean OnBorder( bpix, cip, inp )
        BorPix          bpix;   /* an edge pixel of a node */
        long int        cip;    /* current input position */
        Qt              *inp;   /* input quadtree */
/*-------------------------------------------------------------
 * Determine whether bpix (part of node edge of inp->node[cip])
 * is on the border of quadtree inp
 */
{
        DirType         dir;    /* direction */
        snt_type        white;  /* white snt */
        snt_type        black;  /* black snt */

        switch( bpix.sb ) {
                case S1:
                        dir = SOUTH;
                        white = S1_WHITE;
                        black = S1_BLACK;
                        return OnABorder( bpix, cip, inp, dir, white,
                                                        black );
                case W1:
                        dir = WEST;
                        white = W1_WHITE;
                        black = W1_BLACK;
                        return OnABorder( bpix, cip, inp, dir, white,
                                                        black );
                case SW:
                        dir = SOUTH;
                        white = S1_WHITE;
                        black = S1_BLACK;
                        if ( OnABorder( bpix, cip, inp, dir, white,
                                                        black ) ) {
                                return TRUE;
                        } else {
                                dir = WEST;
                                white = W1_WHITE;
                                black = W1_BLACK;
                                return OnABorder( bpix, cip, inp,
                                        dir, white, black );
                        }
                case S2:
                        dir = SOUTH;
                        white = S2_WHITE;
                        black = S2_BLACK;
                        return OnABorder( bpix, cip, inp, dir, white,
                                                        black );
                case E1:
                        dir = EAST;
                        white = E1_WHITE;
                        black = E1_BLACK;
                        return OnABorder( bpix, cip, inp, dir, white,
                                                        black );
                case SE:
                        dir = SOUTH;
                        white = S2_WHITE;
                        black = S2_BLACK;
```

```
                        if ( OnABorder( bpix, cip, inp, dir, white,
                                        black ) ) {
                                return TRUE;
                        } else {
                                dir = EAST;
                                white = E1_WHITE;
                                black = E1_BLACK;
                                return OnABorder( bpix, cip, inp,
                                        dir, white, black );
                        }
                case W2:
                        dir = WEST;
                        white = W2_WHITE;
                        black = W2_BLACK;
                        return OnABorder( bpix, cip, inp, dir, white,
                                        black );
                case N1:
                        dir = NORTH;
                        white = N1_WHITE;
                        black = N1_BLACK;
                        return OnABorder( bpix, cip, inp, dir, white,
                                        black );
                case NW:
                        dir = WEST;
                        white = W2_WHITE;
                        black = W2_BLACK;
                        if ( OnABorder( bpix, cip, inp, dir, white,
                                        black ) ) {
                                return TRUE;
                        } else {
                                dir = NORTH;
                                white = N1_WHITE;
                                black = N1_BLACK;
                                return OnABorder( bpix, cip, inp,
                                        dir, white, black );
                        }
                case E2:
                        dir = EAST;
                        white = E2_WHITE;
                        black = E2_BLACK;
                        return OnABorder( bpix, cip, inp, dir, white,
                                        black );
                case N2:
                        dir = NORTH;
                        white = N2_WHITE;
                        black = N2_BLACK;
                        return OnABorder( bpix, cip, inp, dir, white,
                                        black );
                case NE:
                        dir = EAST;
                        white = E2_WHITE;
                        black = E2_BLACK;
                        if ( OnABorder( bpix, cip, inp, dir, white,
                                        black ) ) {
                                return TRUE;
                        } else {
                                dir = NORTH;
                                white = N2_WHITE;
                                black = N2_BLACK;
                                return OnABorder( bpix, cip, inp,
                                        dir, white, black );
                        }
        }
}

void SWBorder( cip, inp, outp, quarter_size )
        long int        cip;            /* current input position */
        Qt              *inp;           /* input quadtree */
        Qt              *outp;          /* output quadtree */
        long int        quarter_size;   /* 1/4 # of edge pixels */
/*-------------------------------------------------------------
 * Find the south-west border pixels of the current non-pixel
 * node inp->node[cip] (whose sub-neighbour types have been set)
```



```
 * and store them in outp
 */
{
        int             gf;             /* grouping factor of
                                                this node */
        QtNode          oq;             /* output pixel */
        long int        diff_size;      /* size of this
                                                diff_border_LUT */
        long int        i;              /* counter */
        BorPix          bpix;           /* an edge pixel of
                                                the node */

        if ( MATCH_COLOR(inp->node[cip].snt,S1_BLACK,S1_BLACK) &&
             MATCH_COLOR(inp->node[cip].snt,W1_BLACK,W1_BLACK) ) {
                /* internal */
                return;
        } else {
                gf = inp->node[cip].gf;
                oq.lev = res;
                oq.gf = 0;
                if ( diff_border_LUT[gf] == NULL ) {
                        /* the first time a node of this
                         * gf is encountered
                         */
                        InitDiffBorderLUT( gf, &diff_size );
                }
                /* for each pixel on the SW edge of this node
                 * determine whether it is on the border of inp
                 */
                bpix.loc = inp->node[cip].loc;
                bpix.sb = SW;
                if ( OnBorder( bpix, cip, inp ) ) {
                        oq.loc = bpix.loc;
                        UpdateOutputQt( outp, &oq );
                }
                for ( i = 1; i < quarter_size; i++ ) {
                        bpix.loc += diff_border_LUT[gf][i-1].loc;
                        bpix.sb = diff_border_LUT[gf][i].sb;
                        if ( OnBorder( bpix, cip, inp ) ) {
                                oq.loc = bpix.loc;
                                UpdateOutputQt( outp, &oq );
                        }
                }
        }
}

void SEBorder( cip, inp, outp, quarter_size )
        long int        cip;            /* current input position */
        Qt              *inp;           /* input quadtree */
        Qt              *outp;          /* output quadtree */
        long int        quarter_size;   /* 1/4 # of edge pixels */
/*-------------------------------------------------------------
 * Find the south-east border pixels of the current non-pixel
 * node inp->node[cip] (whose sub-neighbour types have been set)
 * and store them in outp
 */
{
        int             gf;             /* grouping factor of
                                                this node */
        QtNode          oq;             /* output pixel */
        long int        diff_size;      /* size of this
                                                diff_border_LUT */
        long int        i;              /* counter */
        BorPix          bpix;           /* an edge pixel
                                                of the node */

        if ( MATCH_COLOR(inp->node[cip].snt,S2_BLACK,S2_BLACK) &&
             MATCH_COLOR(inp->node[cip].snt,E1_BLACK,E1_BLACK) ) {
                /* internal */
                return;
        } else {
                gf = inp->node[cip].gf;
                oq.lev = res;
```

```
                oq.gf = 0;
                if ( diff_border_LUT[gf] == NULL ) {
                        /* the first time a node of this
                         * gf is encountered
                         */
                        InitDiffBorderLUT( gf, &diff_size );
                }
                /* for each pixel on the SE edge of this node
                 * determine whether it is on the border of inp
                 */
                bpix.loc = inp->node[cip].loc + pow4[res-gf+1];
                bpix.sb = S2;
                if ( OnBorder( bpix, cip, inp ) ) {
                        oq.loc = bpix.loc;
                        UpdateOutputQt( outp, &oq );
                }
                for ( i = quarter_size+1; i < 2*quarter_size; i++ ) {
                        bpix.loc += diff_border_LUT[gf][i-1].loc;
                        bpix.sb = diff_border_LUT[gf][i].sb;
                        if ( bpix.sb == E1W2 ) {
                                /* the last pixel on SE edge */
                                bpix.sb = E1;
                        }
                        if ( OnBorder( bpix, cip, inp ) ) {
                                oq.loc = bpix.loc;
                                UpdateOutputQt( outp, &oq );
                        }
                }
        }
}

void NWBorder( cip, inp, outp, quarter_size )
        long int        cip;            /* current input position */
        Qt              *inp;           /* input quadtree */
        Qt              *outp;          /* output quadtree */
        long int        quarter_size;   /* 1/4 # of edge pixels */
/*-------------------------------------------------------------
 * Find the north-west border pixels of the current non-pixel
 * node inp->node[cip] (whose sub-neighbour types have been set)
 * and store them in outp
 */
{
        int             gf;             /* grouping factor of
                                                this node */
        QtNode          oq;             /* output pixel */
        long int        diff_size;      /* size of this
                                           diff_border_LUT */
        long int        i;              /* counter */
        BorPix          bpix;           /* an edge pixel of
                                           the node */

        if ( MATCH_COLOR(inp->node[cip].snt,N1_BLACK,N1_BLACK) &&
             MATCH_COLOR(inp->node[cip].snt,W2_BLACK,W2_BLACK) ) {
                /* internal */
                return;
        } else {
                gf = inp->node[cip].gf;
                oq.lev = res;
                oq.gf = 0;
                if ( diff_border_LUT[gf] == NULL ) {

                        /* the first time a node of this
                         * gf is encountered
                         */
                        InitDiffBorderLUT( gf, &diff_size );
                }
                /* for each pixel on the NW edge of this node
                 * determine whether it is on the border of inp
                 */
                bpix.loc = inp->node[cip].loc + 2*pow4[res-gf+1];
                bpix.sb = W2;
                if ( OnBorder( bpix, cip, inp ) ) {
                        oq.loc = bpix.loc;
```



```
                        UpdateOutputQt( outp, &oq );
                }
                for ( i = 2*quarter_size+1; i < 3*quarter_size; i++) {
                        bpix.loc += diff_border_LUT[gf][i-1].loc;
                        bpix.sb = diff_border_LUT[gf][i-1].sb;
                        if ( OnBorder( bpix, cip, inp ) ) {
                                oq.loc = bpix.loc;
                                UpdateOutputQt( outp, &oq );
                        }
                }
        }
}

LocCode FirstPixOfE2( cip, inp )
        long int        cip;            /* current input position */
        Qt              *inp;           /* input quadtree */
/*-------------------------------------------------------------
 * Calculate the first pixel of E2 edge of the input node
 * inp->node[cip]
 */
{
        int     gf;             /* grouping factor of this node */
        LocCode first_of_W2;    /* the 1st pixel of W2 edge */
        QtNode  last_of_E1;     /* the last pixel of E1 edge */
        DirType dir;
        QtNode  first_of_E2;

        gf = inp->node[cip].gf;
        first_of_W2 = inp->node[cip].loc + 2*pow4[res-gf+1];
        last_of_E1.loc = first_of_W2 - 1;
        last_of_E1.lev = res;
        last_of_E1.gf = 0;
        /* the NORTH neighbour of last_of_E1 will
                be the first_of_E2 */
        dir = NORTH;
        first_of_E2 = Neighbour( last_of_E1, dir );
        return first_of_E2.loc;
}

void NEBorder( cip, inp, outp, quarter_size )
        long int        cip;            /* current input position */
        Qt              *inp;           /* input quadtree */
        Qt              *outp;          /* output quadtree */
        long int        quarter_size;   /* 1/4 # of edge pixels */
/*-------------------------------------------------------------
 * Find the north-east border pixels of the current non-pixel
 * node inp->node[cip] (whose sub-neighbour types have been set)
 * and store them in outp
 */
{
        int             gf;             /* grouping factor of
                                                this node */
        QtNode          oq;             /* output pixel */
        long int        diff_size;      /* size of this
                                           diff_border_LUT */
        long int        i;              /* counter */
        BorPix          bpix;           /* an edge pixel
                                           of the node */
        LocCode         mask;           /* mask to find SE corner */

        if ( MATCH_COLOR(inp->node[cip].snt,N2_BLACK,N2_BLACK) &&
             MATCH_COLOR(inp->node[cip].snt,E2_BLACK,E2_BLACK) ) {
                /* internal */
                return;
        } else {
                gf = inp->node[cip].gf;
                oq.lev = res;
                oq.gf = 0;
                if ( diff_border_LUT[gf] == NULL ) {

                        /* the first time a node of this
                         * gf is encountered
                         */
```

```
                        InitDiffBorderLUT( gf, &diff_size );
                }
                /* for each pixel on the NE edge of this node
                 * determine whether it is on the border of inp
                 */
                bpix.loc = FirstPixOfE2( cip, inp );
                bpix.sb = E2;
                if ( OnBorder( bpix, cip, inp ) ) {
                        oq.loc = bpix.loc;
                        UpdateOutputQt( outp, &oq );
                }
                for ( i = 3*quarter_size+1; i < 4*quarter_size; i++) {
                        bpix.loc += diff_border_LUT[gf][i-1].loc;
                        bpix.sb = diff_border_LUT[gf][i-1].sb;
                        if ( OnBorder( bpix, cip, inp ) ) {
                                oq.loc = bpix.loc;
                                UpdateOutputQt( outp, &oq );
                        }
                }
        }
}

void NonTrivialBorder( cip, inp, outp )
        long int        cip;    /* current input position */
        Qt              *inp;   /* input quadtree */
        Qt              *outp;  /* output quadtree */
/*-------------------------------------------------------------
 * Find the border pixels of the current non-trivial (meaning
 * node larger than 2x2) node inp->node[cip] (whose sub-neighbour
 * types have been set) and store them in outp
 */
{
        long int        quarter_size;   /* 1/4 # of node
                                                edge pixels */

        quarter_size = (1l << (inp->node[cip].gf)) - 1;
        SWBorder( cip, inp, outp, quarter_size );
        SEBorder( cip, inp, outp, quarter_size );
        NWBorder( cip, inp, outp, quarter_size );
        NEBorder( cip, inp, outp, quarter_size );
}

void TwoByTwoBorder( cip, inp, outp )
        long int        cip;    /* current input position */
        Qt              *inp;   /* input quadtree */
        Qt              *outp;  /* output quadtree */
/*-------------------------------------------------------------
 * Find the border pixels of the current 2x2 node inp->node[cip]
 * (whose sub-neighbour types have been set) and store them in outp
 */
{
        Boolean sw_onborder;    /* SW corner is on border */
        Boolean se_onborder;    /* SE corner is on border */
        Boolean nw_onborder;    /* NW corner is on border */
        Boolean ne_onborder;    /* NE corner is on border */
        QtNode  oq;             /* output pixel */

        /* SW */
        if ( MATCH_COLOR(inp->node[cip].snt,S1_BLACK,S1_BLACK) &&
             MATCH_COLOR(inp->node[cip].snt,W1_BLACK,W1_BLACK) ) {
                sw_onborder = FALSE;
        } else {
                sw_onborder = TRUE;
        }
        /* SE */
        if ( MATCH_COLOR(inp->node[cip].snt,S2_BLACK,S2_BLACK) &&
             MATCH_COLOR(inp->node[cip].snt,E1_BLACK,E1_BLACK) ) {
                se_onborder = FALSE;
        } else {
                se_onborder = TRUE;
        }
        /* NW */
        if ( MATCH_COLOR(inp->node[cip].snt,N1_BLACK,N1_BLACK) &&
```



```
             MATCH_COLOR(inp->node[cip].snt,W2_BLACK,W2_BLACK) ) {
                nw_onborder = FALSE;
        } else {
                nw_onborder = TRUE;
        }
        /* NE */
        if ( MATCH_COLOR(inp->node[cip].snt,N2_BLACK,N2_BLACK) &&
             MATCH_COLOR(inp->node[cip].snt,E2_BLACK,E2_BLACK) ) {
                ne_onborder = FALSE;
        } else {
                ne_onborder = TRUE;
        }
        if ( sw_onborder && se_onborder
                && nw_onborder && ne_onborder ) {
                /* all four pixels are on border, output as a node
                 * to avoid condensation
                 */
                oq.lev = res - 1;
                oq.gf = 1;
                oq.loc = inp->node[cip].loc;
                UpdateOutputQt( outp, &oq );
        } else {
                /* check each pixel in turn */
                oq.lev = res;
                oq.gf = 0;
                if ( sw_onborder ) {
                        oq.loc = inp->node[cip].loc;
                        UpdateOutputQt( outp, &oq );
                }
                if ( se_onborder ) {
                        oq.loc = inp->node[cip].loc + 1;
                        UpdateOutputQt( outp, &oq );
                }
                if ( nw_onborder ) {
                        oq.loc = inp->node[cip].loc + 2;
                        UpdateOutputQt( outp, &oq );
                }
                if ( ne_onborder ) {
                        oq.loc = inp->node[cip].loc + 3;
                        UpdateOutputQt( outp, &oq );
                }
        }
}

void BorderFind( cip, inp, outp )
        long int        cip;    /* current input position */
        Qt              *inp;   /* input quadtree */
        Qt              *outp;  /* output quadtree */
/*-------------------------------------------------------------
 * Find the border pixels of the current node inp->node[cip] and
 * store them in outp
 */
{
        DirType         dir;    /* direction */
        QtNode          oq;     /* output pixel */

        /* set sub-neighbour types in each directions */
        if ( MATCH_COLOR(inp->node[cip].snt,E1_BLACK,UNSET) ) {
                dir = EAST;
                Setsnt( dir, cip, inp );
        }
        if ( MATCH_COLOR(inp->node[cip].snt,S1_BLACK,UNSET) ) {
                dir = SOUTH;
                Setsnt( dir, cip, inp );
        }
        if ( MATCH_COLOR(inp->node[cip].snt,W1_BLACK,UNSET) ) {
                dir = WEST;
                Setsnt( dir, cip, inp );
        }
        if ( MATCH_COLOR(inp->node[cip].snt,N1_BLACK,UNSET) ) {
                dir = NORTH;
                Setsnt( dir, cip, inp );
        }
```

```
        /* output border pixels depending on sub-neighbour types */
        if ( inp->node[cip].lev == res ) {      /* pixel */
                if ( ALL_BLACK( inp->node[cip].snt ) ) {
                        /* internal pixel */
                        return;
                } else {
                        /* border pixel */
                        oq.lev = res;
                        oq.gf = 0;
                        oq.loc = inp->node[cip].loc;
                        UpdateOutputQt( outp, &oq );
                }
        } else if ( inp->node[cip].gf == 1 ) {  /* 2x2 node */
                TwoByTwoBorder( cip, inp, outp );
        } else {
                NonTrivialBorder( cip, inp, outp );
        }
}

main( argn, argc )
        int     argn;
        char    *argc[];
/*-------------------------------------------------------
 */
{
        FILE            *in_fp;         /* input file pointer */
        FILE            *out_fp;        /* output file pointer */
        char            in_fn[64];      /* input file name */
        char            out_fn[64];     /* output file name */
        Qt              inp;            /* input quadtree */
        Qt              outp;           /* output quadtree */
        long int        cur_ipos;       /* current input position */
                                        /* for timing */
        struct tms      tmsstart, tmsend;

        /* check command line */
        CheckNumArg( argn, 3,
                "newborder <input file> <output file>" );
        strcpy( in_fn, argc[1] );
        strcpy( out_fn, argc[2] );
        CheckInfile( &in_fp, in_fn );
        CheckOutfile( &out_fp, out_fn );
        if ( times( &tmsstart ) == -1 ) {       /* start timing */
                fprintf( stderr, "\ntimes() error\n" );
                exit( -4 );
        }
        Init( &inp, &outp, in_fp, out_fp );
        for ( cur_ipos = 0; cur_ipos < inp.size; cur_ipos++ ) {

                BorderFind( cur_ipos, &inp, &outp );
        }
        if ( times( &tmsend ) == -1 ) {         /* end timing */
                fprintf( stderr, "\ntimes() error\n" );
                exit( -4 );
        }
        PrTimes( &tmsstart, &tmsend );          /* print timing */
        PrtOutputQt( &outp, out_fp );
        CleanBLUT();
        ClosePow4();
        CloseIO( &inp, &outp, in_fp, out_fp );
}


/***************************** node-border.c ************************
 * Utility routines for generating the edge pixels of a quadtree
 * node
 * User should call, in sequence, InitBorder(), GenNodeBorder(),
 * CleanBorder(), CleanBLUT().
 * Date         By              Comment
 * ----         --              -------
 * 1994Mar15    S.K.Tang        Created
 */
#include <stdio.h>
```



```
#include "../header/qutil.h"

/* externals */
extern  long int        res;    /* resolution */
extern  LocCode         QtEncode();

/* globals */
LocCode *border_LUT;            /* differences of loc. codes
                                        along x axis */
BorPix  **diff_border_LUT;      /* differences of loc. codes
                                        of node edge */

void InitBorder()
/*-------------------------------------------------------------------
 * Allocate memory and initialize border_LUT and diff_border_LUT
 * Note: global res must be initialized
 */
{
        long int        size;
        long int        i;
        Point           tmpp;
        LocCode         current, last;  /* current and last
                                                encoded x */

        /* allocate border_LUT */
        size = (1l << (res-1)) -1;
        border_LUT = (LocCode *)calloc( size, sizeof(LocCode) );
        if ( border_LUT == NULL ) {
                MemErr( "border_LUT" );
        }
        /* init border_LUT */
        last = 0;
        for ( i = 0; i < size; i++ ) {
                tmpp.v[0] = i+1; tmpp.v[1] = 0;
                current = QtEncode( tmpp );
                border_LUT[i] = current - last;
                last = current;
        }
        /* allocate diff_border_LUT */
        size = res + 1;         /* diff_border_LUT[0] is unused
                                        since gf=0 means pixel */
        diff_border_LUT = (BorPix **)calloc( size, sizeof(BorPix *) );
        if ( diff_border_LUT == NULL ) {
                MemErr( "diff_border_LUT" );
        }
        /* init diff_border_LUT */
        for ( i = 0; i < size; i++ ) {
                diff_border_LUT[i] = NULL;
        }
}

void InitDiffBorderLUT( gf, diff_size )
        int             gf;             /* grouping factor */
        long int        *diff_size;     /* size of this LUT */
/*-------------------------------------------------------------------
 * Initialize diff_border_LUT[gf] for node of grouping factor gf.
 *
 * Note: InitBorder() should have been invoked
 */
{
        long int        size;                   /* # of differences
                                                   for this gf */
        long int        r,last;                 /* running index */
        long int        j,m;
        int             i;

        /* avoid creating garbage */
        if ( diff_border_LUT[gf] != NULL ) {
                free( diff_border_LUT[gf] );
        }
        /* special cases if gf=0 or gf=1 */
        if ( gf == 0 ) {
                *diff_size = 0;
```

```
        return;
    }
    if ( gf == 1 ) {
        diff_border_LUT[gf] = (BorPix *)calloc( 3,
                                        sizeof(BorPix) );
        if ( diff_border_LUT[gf] == NULL ) {
            fprintf( stderr, "\nFor gf=%d\n", gf );
            MemErr( "diff_border_LUT[gf]" );
        }
        diff_border_LUT[gf][0].loc = 1;
        diff_border_LUT[gf][0].sb = SW;
        diff_border_LUT[gf][1].loc = 1;
        diff_border_LUT[gf][1].sb = E1W2;
        diff_border_LUT[gf][2].loc = 1;
        diff_border_LUT[gf][2].sb = NE;
        *diff_size = 3;
        return;
    }
    /* allocate memory for LUT of this gf (see 3/11/94 research
                                        note) */
    size = 2 * ((1l << (gf+1)) - 3) + 1;
    diff_border_LUT[gf] = (BorPix *)calloc( size,
                                        sizeof(BorPix) );
    if ( diff_border_LUT[gf] == NULL ) {
        fprintf( stderr, "\nFor gf=%d\n", gf );
        MemErr( "diff_border_LUT[gf]" );
    }
    /* for the lower south and west edge */
    diff_border_LUT[gf][0].loc = 1;
    diff_border_LUT[gf][0].sb = SW;
    r = 1;
    for ( i = 0; i <= gf-2; i++ ) {
        j = 1l << i;
        last = r;
        for ( m = 0; m < j; m++ ) {
            diff_border_LUT[gf][r].loc = border_LUT[m];
            diff_border_LUT[gf][r].sb = S1;
            r++;
        }
        for ( m = 0; m < j; m++ ) {
            diff_border_LUT[gf][r].loc =
                    2 * diff_border_LUT[gf][last+m].loc;
            diff_border_LUT[gf][r].sb = W1;
            r++;
        }
    }
    /* for the lower south and east edge */
    j = (1l << (gf-1)) - 1;
    last = r;
    for ( m = 0; m < j; m++ ) {
        diff_border_LUT[gf][r].loc = border_LUT[m];
        diff_border_LUT[gf][r].sb = S2;
        r++;
    }
    diff_border_LUT[gf][r].loc =
                    2 * diff_border_LUT[gf][last].loc;
    diff_border_LUT[gf][r].sb = SE;
    r++;
    for ( m = 1; m < j; m++ ) {
        diff_border_LUT[gf][r].loc =
                    2 * diff_border_LUT[gf][last+m].loc;
        diff_border_LUT[gf][r].sb = E1;
        r++;
    }
    /* just between lower and upper edge */
    diff_border_LUT[gf][r].loc = 1;
    diff_border_LUT[gf][r].sb = E1W2;
    r++;
    /* for upper edge */
    for ( m = size/2-1; m >= 0; m-- ) {
        diff_border_LUT[gf][r].loc =
                    diff_border_LUT[gf][m].loc;
        diff_border_LUT[gf][r].sb = SIZE_OF_SubBorderType -
```



```
                    diff_border_LUT[gf][m].sb;
            r++;
        }
        *diff_size = r;
    }

    void GenNodeBorder( q, bp, size )
        QtNode          q;      /* a quadtree node */
        BorPix          **bp;   /* addr of pointer to the
                                   edge pixels of q */
        long int        *size;  /* ptr to # of edge pixels */
    /*-----------------------------------------------------------------
     * Given a node q, return its edge pixels through bp,
     * along with size
     * Note: InitBorder() should be invoked first
     * Note: on entry bp should be NULL; user should call CleanBorder()
     *       after using bp
     */
    {
        int             gf;             /* group factor */
        long int        half_size;      /* half # of edge pixels  */
        long int        diff_size;      /* size of this diff LUT */
        long int        i;

        gf = q.gf;
        /* avoid creating garbage */
        if ( *bp != NULL ) {
            free( *bp );
        }
        /* if q is a pixel */
        if ( gf == 0 ) {
            *size = 1 ;
            *bp = (BorPix *)calloc( 1, sizeof(BorPix) );
            if ( *bp == NULL ) {
                MemErr( "*bp" );
            }
            (*bp)[0].loc = q.loc;
            (*bp)[0].sb = E1W2;
            return;
        }
        /* q is not a pixel, allocate memory to store the edge */
        half_size = (1l << (gf+1)) - 2;
        *size = 2 * half_size;
        *bp = (BorPix *)calloc( *size, sizeof(BorPix) );
        if ( *bp == NULL ) {
            MemErr( "*bp" );
        }
        /* if first time node of gf size is encountered */
        if ( diff_border_LUT[gf] == NULL ) {
            InitDiffBorderLUT( gf, &diff_size );
        }
        /* generate the lower edge */
        (*bp)[0].loc = q.loc;
        (*bp)[0].sb = diff_border_LUT[gf][0].sb;
        for ( i = 1; i < half_size; i++ ) {
            (*bp)[i].loc = diff_border_LUT[gf][i-1].loc +
                                    (*bp)[i-1].loc;
            (*bp)[i].sb = diff_border_LUT[gf][i].sb;
        }
        (*bp)[half_size-1].sb = E1;
        /* generate the upper edge */
        (*bp)[half_size].loc = (*bp)[half_size-1].loc + 1;
        (*bp)[half_size].sb = W2;
        for ( i = half_size+1; i < *size; i++ ) {
            (*bp)[i].loc = diff_border_LUT[gf][i-1].loc +
                                    (*bp)[i-1].loc;
            (*bp)[i].sb = diff_border_LUT[gf][i-1].sb;
        }
    }

    void CleanBorder( bp )
        BorPix  *bp;    /* pointer to the edge pixels of q */
    /*-----------------------------------------------------------------
```

```
 * Free memory used by bp
 */
{
        free( bp );
}

void CleanBLUT()
/*--------------------------------------------------------------------
 * Free memory used by LUT
 *
 * Note: global res must be initialized
 */
{
        int     i;

        free( border_LUT );
        for ( i = 0; i <= res; i++ ) {
                free( diff_border_LUT[i] );
        }
        free( diff_border_LUT );
}


/*********************************************************************
*** quad.c  (qdsv1p-ha.h)                    93/02/18      GFS    ***
*** A collection of relevant procedures for quadtree programs.     ***
*** Method used for encoding: two-part double-vector encoding,     ***
***                 for decoding: one-part single-vector hash-decoding.***
*********************************************************************
  This file contains encoding and decoding procedures for
  the quad domains of resolution  2 <= res <= 12.
  NOTES:
     (1) The variable 'res' to hold the resolution of the quad-domain
         is declared in this file. It must be defined (e.g., by
         reading it in).  It is the only global variable.
     (2) After 'res' has been defined, invoke the
         procedure 'initialise' by: initialise();
     (3) All variables have been declared to be of data
         type 'long int'; in particular, declare the coordinates
         and the location code as datatypes 'long int'.
     (4) In the body of the application program, use the following
         procedures.
         For decoding, use the procedure call
            decode(loc, &x, &y);
         for encoding, use the procedure call
            loc = encode(x,y);
         The algorithms assume and check that the
         coordinates x, y are not negative (i.e., both x and y
         must be positive or zero).
         If it is desired that negative coordinates are encoded,
         use the following formula (which will result in
         wrap-around of the coordinates):
           long int mask2;
           mask2 = (1 << 2) - 1;
           loc = encode(x & mask2, y & mask2);
     (5) For timing purposes, calls to encode and decode should be
         replaced by in-line statements by copying the statements
         in the body of these two procedures.
   ------------------------------------------------------------------ */
#define  RHAT   6
#define  TRUE   1
#define  FALSE  0
#include <stdio.h>

void exit(int status);
/* void *calloc(size_t nitems, size_t size); */

      long int  res;
static long int  tx, tn, k2, tlpx, kshx;
static long int *qevx, *qevy, *qdvx;

long int qdilate(long int k)
{
```



```
   long int i, t;

   t = k & 1;
   for (i = 1; i < res; ++i)
      t |= ((k & (1<<i)) << i);
   return t;
}

long int encode(long int i,long int j)
{
    /* static external long int qevx, qevy, RHAT, tn, k2; */
    return (   (qevx[i & tn ] | qevy[j & tn ])
            | ((qevx[i>>RHAT] | qevy[j>>RHAT]) << k2));
}

void decode(long int loc, long int *x, long int *y)
{
   /* static external long int qdvx, tx, tlpx, kshx */
   *x  = qdvx[ (loc       & tlpx) | ( (loc       & tx) >> kshx)];
   *y  = qdvx[((loc >> 1) & tlpx) | (((loc >> 1) & tx) >> kshx)];
}

void initialise(void)
{
   long int m, i, restmp;
   if (res < 2)
   {
      printf(" ERROR: resolution is too low. Initialise the");
      printf(" variable 'res' before\n    invoking 'initialise'\n");
      exit(1);
   }
   if (res > 2*RHAT)
   {
      printf(" ERROR: resolution is too high! ");
      printf(" Max resolution is res = %ld\n",(2*RHAT));
      exit (1);
   }
   restmp = res;       /* temporary redefinition of res in order to */
   res  =  RHAT;       /* invoke qdilate for initialisation of the  */
   tn   = (1l<<RHAT);  /*                          the encode vectors */
   qevx = (long int *) calloc(tn, sizeof(long int));
   qevy = (long int *) calloc(tn, sizeof(long int));
   tn   =  tn - 1l;
   for (i = 0; i <= tn; ++i)
   {
      qevx[i] = qdilate(i);
      qevy[i] = qevx[i] << 1;
   }
   res  = restmp;                        /* reset resolution */
   k2   = RHAT << 1;                     /* constants */
   tx   = qdilate((1l << res) - 1l);
   tlpx = qdilate((1l << ((res + 1l) / 2l)) - 1l);
   if (res & 1l)
      kshx = res;
   else
      kshx = res - 1l;
   qdvx   = (long int *) calloc((1 << res), sizeof(long int));
   for (i = 0; i < (1l << res); ++i)
   {
      m = qdilate(i);
      qdvx[(m & tlpx) | (m >> kshx)] = i;
   }
}


/****************************** qutil.c ********************************
 * Utility routines for quadtree-related problems
 * Date         By              Comment
 * ----         --              -------
 * 1994Feb11    S.K.Tang        Created
 */
#include <stdio.h>
#include "../header/qutil.h"
```

```
/* externals */
extern  long int        res;            /* resolution */

/* globals */
LocCode ncode_LUT[8];   /* neighbour code lookup table */
LocCode *pow4;          /* pow4[i] = 4^(res-i) */
LocCode *pow4m1;        /* pow4m1[i] = pow4[i]-1 */
LocCode Tx;             /* mask for dilated x */
LocCode Ty;             /* mask for dilated y */

/******************************** I/O ******************************/
void GetInputQt( inp, in_fp )
        Qt              *inp;           /* input quadtree */
        FILE            *in_fp;         /* input file pointer */
/*-----------------------------------------------------------------
 * Allocate and get input quadtree
 * Note: initialize global res in this routine
 */
{
        QtNode          tn;             /* a quadtree node */
        long int        inp_alloc;      /* allocated size */
        long int        inps;           /* running input size */

        /* read resolution */
        fscanf( in_fp, "%ld", &res );
        /* allocate memory for input qt */
        inp_alloc = DEFAULT_NUM_IN_NODES;
        inp->node = (QtNode *)calloc( inp_alloc, sizeof(QtNode) );
        if ( inp->node == NULL ) {
                MemErr( "inp" );
        }
        inps = 0;
        /* read input qt and calculate input size */
        while ( fscanf( in_fp, "%ld %d", &tn.loc, &tn.lev ) != EOF ) {
                tn.gf = res - tn.lev;
                if ( inps >= inp_alloc ) {
                        /* input size is larger than allocated size */
                        inp_alloc <<= 1;/* try twice as much memory */
                        inp->node = (QtNode *)
                                realloc( inp->node,
                                        inp_alloc*sizeof(QtNode) );
                        if ( inp->node == NULL ) {
                                MemErr( "inp again" );
                        }
                }
                inp->node[inps].loc = tn.loc;
                inp->node[inps].lev = tn.lev;
                inp->node[inps].gf = tn.gf;
                inp->node[inps].snt = UNSET;
                inps++;
        }
        /* free unnecessary memory for input qt */
        if ( inps < inp_alloc ) {
                inp->node = (QtNode *)
                        realloc( inp->node, inps*sizeof(QtNode) );
                if ( inp->node == NULL ) {
                        MemErr( "inp just fit" );
                }
        }
        inp->size = inps;
        inp->allocated = inps;
}

void AllocOutputQt( outp, out_fp )
        Qt              *outp;          /* output quadtree */
        FILE            *out_fp;        /* output file pointer */
/*-----------------------------------------------------------------
 * Allocate output quadtree
 */
{
        long int        outp_alloc;     /* allocated size */
```



```
        /* allocate default memory */
        outp_alloc = DEFAULT_NUM_OUT_NODES;
        outp->node = (QtNode *) calloc( outp_alloc, sizeof(QtNode) );
        if ( outp->node == NULL ) {
                MemErr( "outp" );
        }
        /* init current size to 0 */
        outp->size = 0;
        outp->allocated = outp_alloc;
}

void UpdateOutputQt( outp, q )
        Qt      *outp;  /* output quadtree */
        QtNode  *q;     /* node to be stored */
/*-----------------------------------------------------------------
 * Update output quadtree by storing a node q into it
 */
{
        long int        outp_alloc;     /* # of node allocated */

        outp_alloc = outp->allocated;
        if ( outp->size >= outp_alloc ) {
                /* output quadtree is larger than
                        current memory allocated */
                outp_alloc <<= 1;/* try twice as much memory */
                outp->node = (QtNode *) realloc( outp->node,
                                        outp_alloc*sizeof(QtNode) );
                if ( outp->node == NULL ) {
                        MemErr( "outp again" );
                }
                outp->allocated = outp_alloc;
        }
        outp->node[outp->size].loc = q->loc;
        outp->node[outp->size].lev = q->lev;
        outp->node[outp->size].gf = q->gf;
        (outp->size)++;
}

void PrtOutputQt( outp, out_fp )
        Qt              *outp;          /* output quadtree */
        FILE            *out_fp;        /* output file pointer */
/*-----------------------------------------------------------------
 * Print output quadtree
 * Note: global res must be initialized
 */
{
        long int        ii;

        /* print resolution */
        fprintf( out_fp, "%ld\n", res );
        /* print nodes */
        for ( ii = 0; ii < outp->size; ii++ ) {
                fprintf( out_fp, "%ld %d\n",
                        outp->node[ii].loc, outp->node[ii].lev );
        }
}

void CloseIO( inp, outp, in_fp, out_fp )
        Qt      *inp;   /* input quadtree */
        Qt      *outp;  /* output quadtree */
        FILE    *in_fp; /* input file pointer */
        FILE    *out_fp;/* output file pointer */
/*-----------------------------------------------------------------
 * Free memory related to I/O
 */
{
        free( inp->node );
        free( outp->node );
        fclose( in_fp );
        fclose( out_fp );
}

void Prtsnt( cip, inp )
```

```
        long int        cip;    /* current input node position */
        Qt              *inp;   /* input quadtree */
/*-------------------------------------------------------------------
 * Print the sub-neighbour type for current input node
 * This is mainly for debugging, thus output to stdout
 */
{
        printf( "%ld %d 0x%x\n", inp->node[cip].loc,
                        inp->node[cip].lev, inp->node[cip].snt  );
}

/************************** encode/decode ***************************/
void InitPow4()
/*-------------------------------------------------------------------
 * Initialize globals pow4[i] = 4^(res-i) and pow4m1[i] = pow4[i]-1
 */
{
        int     i;              /* loop counter */
        LocCode temp=1l;        /* temporary */

        pow4 = (LocCode *) calloc( (res+1), sizeof(int) );
        if ( pow4 == NULL ) {
                MemErr( "pow4" );
        }
        pow4m1 = (LocCode *) calloc( (res+1), sizeof(int) );
        if ( pow4m1 == NULL ) {
                MemErr( "pow4m1" );
        }
        for ( i = res; i >= 0; --i ) {
                pow4[i] = temp;
                pow4m1[i] = temp - 1;
                temp <<= 2;
        }
}

void ClosePow4()
/*-------------------------------------------------------------------
 * Free memory used by globals pow4 and pow4m1
 */
{
        free( pow4 );
        free( pow4m1 );
}

void InitMask()
/*-------------------------------------------------------------------
 * Initialize globals Tx=01...01 (there are res 1's) and Ty
 * Note: global res must be initialized first
 * Note: InitPow4() must be invoked first
 */
{
        int     i;      /* loop counter */

        Tx = 1l;
        for ( i = res-1; i > 0; i-- ) {
                Tx += pow4[i];
        }
        Ty = Tx << 1;
}

void InitCoding()
/*-------------------------------------------------------------------
 * Initialize LUT for encoding/decoding
 * Use algorithm in quad.c
 * Note: global res must be initialized first
 */
{
        initialise();
}

LocCode QtEncode( pt )
        Point   pt;     /* the cartesian point to be encoded */
/*-------------------------------------------------------------------
```



```
 * Encode a cartesian point into a location code
 * Use algorithm in quad.c
 * Note: InitCoding() must be invoked first
 */
{
        return encode( pt.v[0], pt.v[1] );
}

Point QtDecode( n )
        LocCode n;      /* the location code to be decoded */
/*-------------------------------------------------------------------
 * Decode a location code into a cartesian point
 * Use algoritm in quad.c
 * Note: InitCoding() must be invoked first
 */
{
        Point   p;      /* temporaries */

        decode( n, &(p.v[0]), &(p.v[1]) );
        return p;
}

/************************** arithmetic ****************************/
LocCode LocAdd( n1, n2 )
        LocCode n1;     /* location code 1 */
        LocCode n2;     /* location code 2 */
/*-------------------------------------------------------------------
 * Compute n1 + n2 using location code arithmetic
 *
 * Note: InitMask() must be invoked first
 */
{
        LocCode tmp1,tmp2;      /* temporaries */

        tmp1 = ((n1 | Ty) + (n2 & Tx)) & Tx;
        tmp2 = ((n1 | Tx) + (n2 & Ty)) & Ty;
        return ( tmp1 | tmp2 );
}

LocCode LowerRightPixel( q )
        QtNode  q;
/*-------------------------------------------------------------------
 * Compute the location code of the lower right pixel of node q
 */
{
        LocCode mask;
        int     i;

        mask = 01;
        for ( i = 0; i < q.gf; i++ ) {
                mask = (mask << 2) | 11;
        }

        return (q.loc | mask);
}

LocCode UpperLeftPixel( q )
        QtNode  q;
/*-------------------------------------------------------------------
 * Compute the location code of the upper left pixel of node q
 */
{
        LocCode mask;
        int     i;

        mask = 01;
        for ( i = 0; i < q.gf; i++ ) {
                mask = (mask << 2) | 21;
        }
        return (q.loc | mask);
}

LocCode UpperRightPixel( q )
```

```
        QtNode  q;
/*-------------------------------------------------------------
 * Compute the location code of the upper right pixel of node q
 */
{
        LocCode mask;
        int     i;

        mask = 01;
        for ( i = 0; i < q.gf; i++ ) {
                mask = (mask << 2) | 31;
        }

        return (q.loc | mask);
}

/************************* neighbour ****************************/
void InitNeighbourLUT()
/*-------------------------------------------------------------
 * Set up lookup table for neighbour code
 * Note: global res must be initialized first
 */
{
        long int        mask2;  /* mask for -ve numbers */
        Point           tp;     /* temporary */
        DirType         dir;    /* neighbour direction */
        mask2 = (1 << res) - 1;
        dir = EAST;
        tp.v[0] = 1; tp.v[1] = 0;
        ncode_LUT[dir] = QtEncode( tp );
        dir = NORTH_EAST;
        tp.v[1] = 1;
        ncode_LUT[dir] = QtEncode( tp );
        dir = NORTH;
        tp.v[0] = 0;
        ncode_LUT[dir] = QtEncode( tp );
        dir = NORTH_WEST;
        tp.v[0] = -1 & mask2;
        ncode_LUT[dir] = QtEncode( tp );
        dir = WEST;
        tp.v[1] = 0;
        ncode_LUT[dir] = QtEncode( tp );
        dir = SOUTH_WEST;
        tp.v[1] = -1 & mask2;
        ncode_LUT[dir] = QtEncode( tp );
        dir = SOUTH;
        tp.v[0] = 0;
        ncode_LUT[dir] = QtEncode( tp );
        dir = SOUTH_EAST;
        tp.v[0] = 1;
        ncode_LUT[dir] = QtEncode( tp );
}

QtNode Neighbour( q, dir )
        QtNode  q;      /* the current quad node */
        DirType dir;    /* neighbour direction */
/*-------------------------------------------------------------
 * Calculate neighbour code of equal size using location
 * code addition
 * Note: InitNeighbourLUT() must be invoked first
 */
{
        LocCode delta_n;        /* delta location code to add on */
        QtNode  n;              /* neighbour of q */

        delta_n = ncode_LUT[dir];
        delta_n <<= 2 * q.gf;   /* accounts for
                                   neighbour of same size */
        n.loc = LocAdd( q.loc, delta_n );
        n.lev = q.lev;
        n.gf = q.gf;
        return n;
}
```



```
Boolean NeighbourFinding( n, predecessor, successor, inp, position )
        QtNode          n;      /* the neighbour to be sought after */
        QtNode          **predecessor;/* not found,
                                         return predecessor */
        QtNode          **successor;/* not found, return successor */
        Qt              *inp;           /* the input quadtree */
        long int        *position;      /* index position if found */
/*-------------------------------------------------------------------
 * Find the neighbour n in the input quadtree
 * If found, return TRUE
 * If not found, return FALSE and assign the predecessor and successor
 * as defined in [Yang & Lin, 1990]
 * Note: Use binary search.
 */
{
        long int        l,r;    /* left and right index */
        long int        m;      /* middle index */

        l = 0;
        r = inp->size - 1;
        m = ( l + r ) / 2;
        while ( l < r ) {
                if ( n.loc < inp->node[m].loc ) {
                        r = m;
                } else if ( inp->node[m].loc < n.loc ) {
                        l = m + 1;
                } else {
                        if ( inp->node[m].lev == n.lev ) {
                                /***** FOUND *****/
                                *position = m;
                                return TRUE;
                        } else {
                                break;
                        }
                }
                m = ( l + r ) / 2;
        }
        if ( inp->node[m].loc == n.loc && inp->node[m].lev == n.lev) {
                /* boundary case l=r=m and just found */
                *position = m;
                return TRUE;
        } else if ( inp->node[m].loc <= n.loc &&
                        inp->node[m].lev < n.lev ) {
                *predecessor = &(inp->node[m]);
                *successor = &(inp->node[m+1]);
        } else {
                if ( m > 0 ) {
                        *predecessor = &(inp->node[m-1]);
                } else {
                        *predecessor = &(inp->node[m]);
                }
                *successor = &(inp->node[m]);
        }
        return FALSE;
}

NeiType NeighbourCheck( q, n, dir, inp, position )
        QtNode          q;      /* the current node */
        QtNode          n;      /* the neighbour of q to be checked */
        DirType         dir;    /* direction of n relative to q */
        Qt              *inp;   /* the input quadtree */
        long int        *position;/* index position if n is in inp */
/*-------------------------------------------------------------------
 * Determine the type of the neighbour node n
 */
{
        Boolean         found;  /* whether n is in inp */
        QtNode          *pre;   /* predecessor of n in inp */
        QtNode          *suc;   /* successor of n in inp */
        int             range;  /* temporary */

        /* check whether n is wrapped around */
```

```
        /* i.e. whether q is on the edge of the domain */
        switch ( dir ) {
                case EAST:
                case NORTH:
                        if ( n.loc <= q.loc ) {
                                /* n is wrapped around */
                                return WHITE;
                        }
                        break;
                case SOUTH:
                case WEST:
                        if ( n.loc >= q.loc ) {
                                /* n is wrapped around */
                                return WHITE;
                        }
                        break;
                default:
                        fprintf( stderr,
                                "NeighbourCheck: cannot happen\n" );
                        break;
        }
        /* n is not wrapped around */
        found = NeighbourFinding( n, &pre, &suc, inp, position );
        if ( found ) {
                return BLACK_EXACT;
        }
        if ( n.gf < pre->gf ) {
                range = 1 << (2 * pre->gf);     /* pow(4, pre->gf) */
                if ( (n.loc ^ pre->loc) < range ) {
                        /* pre is super-neighbour of q */
                        return BLACK_NOTEXACT;
                }
        }
        if ( suc->gf < n.gf ) {
                range = 1 << (2 * n.gf);        /* pow(4, n.gf) */
                if ( (suc->loc ^ n.loc) < range ) {
                        /* suc is sub-neighbour of q */
                        return GRAY;
                }
        }
        return WHITE;
}


/****************************** util.c ********************************
 * General utility routines
 * Date         By              Comment
 * ----         --              -------
 * 1994Feb25    S.K.Tang        Created
 * 1994May26    S.K.Tang        Add PrTimes()
 */
#include <stdio.h>
#include <sys/times.h>
#include <unistd.h>

void MemErr( obj )
        char    *obj;   /* pointer to object to be allocated memory */
/*--------------------------------------------------------------------
 * Print memory allocation error
 */
{
        fprintf( stderr, "\nCannot allocate memory for %s\n", obj );
        exit( -1 );
}

void CheckNumArg( argn, n, syntax )
        int     argn;
        int     n;
        char    *syntax;
/*--------------------------------------------------------------------
 * Check number of command line arguments
 */
{
```



```
        if ( argn != n ) {
                fprintf( stderr, "Usage: %s\n", syntax );
                exit( 1 );
        }
}

void CheckInfile( fp, fn )
        FILE    **fp;   /* address of input file pointer */
        char    *fn;    /* input file name */
/*--------------------------------------------------------------------
 * Check existence of input file
 */
{
        *fp = fopen( fn, "r" );
        if ( *fp == NULL ) {
                fprintf( stderr,
                        "Input file \"%s\" does not exist\n", fn );
                exit( 1 );
        }
}

void CheckOutfile( fp, fn )
        FILE    **fp;   /* address of output file pointer */
        char    *fn;    /* output file name */
/*--------------------------------------------------------------------
 * Check existence of output file
 */
{
        *fp = fopen( fn, "r" );
        if ( *fp != NULL ) {
                fprintf( stderr, "file \"%s\" already exist, ", fn );
                fprintf( stderr, "OK to overwrite? (y/n): " );
                if ( getchar() == 'n' ) {
                        fprintf( stderr,
                          "\n\tRe-enter with new output filename\n" );
                        exit( 2 );
                }
        }
        *fp = fopen( fn, "w" );
}

void PrTimes( tmsstart, tmsend )
        struct tms      *tmsstart;      /* starting CPU time */
        struct tms      *tmsend;        /* ending CPU time */
/*--------------------------------------------------------------------
 * Print CPU times.
 * Adapted from "Advanced Programming in the UNIX Environment" by
 * W. Richard Stevens, pp.234, Addison-Wesley 1992
 */
{
        static long     clktck = 0;

        if ( clktck == 0 ) {
                /* fetch clock ticks per second first time */
                if ( (clktck = sysconf(_SC_CLK_TCK)) < 0 ) {
                        fprintf( stderr, "\nError from sysconf" );
                        exit( -4 );
                }
        }
        fprintf( stderr, "\tuser: %7.2f\n",
                (tmsend->tms_utime - tmsstart->tms_utime) /
                        (double) clktck );
        fprintf( stderr, "\tsys: %7.2f\n",
                (tmsend->tms_stime - tmsstart->tms_stime) /
                        (double) clktck );
        fprintf( stderr, "\tchild user: %7.2f\n",
                (tmsend->tms_cutime - tmsstart->tms_cutime) /
                        (double) clktck );
        fprintf( stderr, "\tchild sys: %7.2f\n",
                (tmsend->tms_cstime - tmsstart->tms_cstime) /
                        (double) clktck );
}
```