Filling Contours on the Linear Quadtree Domain by Insertion and Traversal

by
Lars Martin Wilke
B.A.Sc., The University of Waterloo, 1986

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
December 1993

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 21 DEC 1993

Abstract

An algorithm is presented which fills contours described by a set of pixels whose locations are given in unsorted linear quadtree (LQT) form. Each pixel requires an associated blocking code which indicates locally in which direction the region should grow. The technique takes advantage of certain characteristics of LQT location codes to determine if the boundary pixels fall on the edges or vertices of lower resolution LQT cells. This information allows for the automatic determination of boundary cells after the pixels have been inserted into a regular quadtree. Remaining uncoloured cells are coloured using conventional techniques. The algorithm compares favourably with existing LQT filling algorithms both in time and space complexity. The technique avoids explicit sorting of the input pixels or output cells by location code, and no condensation of cells is required. Traversing the regular quadtree in preorder generates the sorted output cells as an LQT.

Table of Contents

List of Tables

List of Figures

Acknowledgments

I would like to thank the following individuals for their support and guidance while completing my Master's thesis. Dr. Schrack, whose enthusiasm and open door policy made work with him a pleasure, and whose funding helped to support my research. Many thanks to Sang Mah, who originally convinced me of the value of returning to university and whose initial encouragement helped me past self doubt. The calm friendship of Angela Berlin was much appreciated, and made my time as a Master's student an enjoyable one. Most importantly I wish to thank my family, whose unquestioning support, both emotional and material, has been with me through all of the endeavors of my life.

1.0 Introduction

1.1 Regular and Linear Quadtrees

Quadtrees are a data structure commonly used to represent objects according to their spatial occupancy on a (2^r x 2^r) raster or quadtree domain. As such, they are commonly used in raster graphics for image compression, and in geographic information systems as a spatial indexing technique. r is the called the *resolution* of the raster and corresponds to the maximum depth of the quadtree. It is normally fixed for a given application.

The quadtree is constructed by dividing the raster (represented by the root node of the quadtree) into four quadrants, and successively dividing each quadrant by four if it is only partially contained by the object. This subdivision continues until either the quadrant is completely contained or not contained, or until the maximum depth of the quadtree has been reached (pixel level).

Each node of the quadtree represents a subquadrant on the raster, and is either black (occupied), white (unoccupied) or grey (further subdivided). The size of the quadrant represented by the node depends on the node-depth in the quadtree.

By labeling each branch of a quadtree node with a quaternary digit, (e.g. 0:SW, 1:SE, 2:NW, 3:NE), a pointerless representation of the quadtree can be created. A quaternary number is formed by concatenating the branch label digits as the quadtree is traversed from the root to a black node. If the black node is not at the pixel level, the least significant digits are assigned the value 0 such that the number of digits is equal to the resolution number of the domain. This quaternary number is called a *location code*, and it is associated with a number indicating the level of the black node inside the quadtree. In practical applications, the location code and level can be stored in a four-byte computer word. The sequence of location codes for all pixels in the quadtree domain is equivalent to the Morton sequence [1] when the location code is given in decimal form.

A sorted list of location codes with associated node levels, representing an object is referred to as a

linear quadtree (LQT). Besides being a more space efficient representation than the regular quadtree, many of the most common operations on objects such as rotation, reflection [2], neighbour finding [3], union and intersection [4], can be performed efficiently on linear quadtrees since explicit traversal of the quadtree can be avoided. Simple manipulations of the location codes are used instead.

Location codes have the very useful property that they can be formed by interleaving the bits of the x and y coordinates of any pixel in the quadtree domain. This property, first pointed out by Gargantini [5], greatly simplifies the task of encoding (insertion into quadtree), decoding, and neighbour finding. It has also lead to a new area of investigation called location code arithmetic [2], which exploits patterns and symmetries of the Morton sequence to derive faster operations on LQT objects.

1.2 Filling in the LQT Domain

Two-dimensional region filling is a problem which is as old as computer graphics itself. The algorithms which have been developed are usually specific to the region representations used, and comparisons between algorithms must be made in this context.

For quadtree representations, many filling algorithms have been developed. They are particularly important since they provide the means for converting between traditional image representations such as polygons or boundary chains to the region quadtree. As quadtrees find wider acceptance in computer cartography and computer graphics, the efficiency of the conversion algorithms will become more critical.

Samet presents an algorithm for regular quadtrees that fills regions whose boundaries are represented by *chain codes* [6, 7]. The algorithm first builds a partial quadtree by inserting boundary pixels as black leaves as they are derived from the chain code. Intermediate nodes in the tree initially have undefined colours (they are called uncoloured) which are determined in the second phase of the algorithm. New boundary pixels are generated in the first phase by using neighbour finding operations as the boundary chain is traversed. A blocking code is produced for each pixel, which stores

information regarding which edges of the pixel are shared with pixels exterior to the filled region. The blocking code consists of four bits, one for each edge of a node. A bit is set if the corresponding edge of the node forms the boundary of the region. The second phase of Samet's algorithm traverses the quadtree, colouring each intermediate node encountered, based on the configuration and blocking codes of the children of the node. Note that the concept of blocking codes defined here is common to other quadtree filling algorithms [8, 9, 10], and is indeed used in the algorithm presented in this thesis.

Algorithms to fill regions whose perimeters are described by pixels located by LQT location codes have been published by Gargantini, Atkinson and Walsh [8, 9] and by Mark and Abel [10]. In the Gargantini algorithms, the input is a sorted list of LQT location codes of the boundary pixels along with their associated blocking codes. The algorithms use the blocking codes to find missing pixels, and later, as the algorithms progress up the virtual quadtree, blocking codes propagated up from lower levels are used to find missing quadrants.

In both of the Gargantini algorithms, the input must be sorted, although in [8] a bucket sort is incorporated as part of the recursive filling routine. Here, the initial pixel list is split into four sublists depending on which quadrant of the image each boundary pixel is in. The procedure is reinvoked for each sublist, which are again subdivided according to the subquadrant of the image that each list element appears in. After r subdivisions for a $2^r \times 2^r$ raster, each sublist contains at most four pixels corresponding to the children of a 2×2 cell in the image. The arrangement of these siblings, along with their blocking codes, can be used to find any missing black siblings. If four black siblings exist, they can be merged to form a single black cell. The resulting sublist of black nodes is passed back as the recursion unwinds, allowing larger and larger missing black nodes to be found, and if possible, larger siblings to be merged. The output of this algorithm is only partially sorted, but by adding a special output data structure, and some minimal algorithmic complexity, the output can be sorted.

The second phase of the algorithm presented by Mark and Abel also takes a sorted list of boundary pixels with blocking codes to derive the region quadtree. The main theorem underlying the second

phase states that, given a sorted list of boundary pixels, the absent pixels between two consecutive pixels in the list must either be all the same colour, or can change colours at most one time. In the second case, two distinctly coloured sets of pixels are derived. The location of the pixel which follows the change of colour between sets can be derived by finding the nearest common ancestor of the two consecutive pixels in the boundary list. The first pixel of one of the quadrants of the ancestor will be that pixel which follows the colour change. The correct quadrant is chosen by examining the blocking codes of the current consecutive boundary pixel pair. The final output of the algorithm is a set of Morton sequence runs of white and black pixels. Translation from runs of pixels to LQT cells is relatively straightforward.

The purpose of this thesis is to present a new region filling algorithm for boundary pixels presented as LQT location codes. The algorithm avoids sorting the input or output location codes, and does not condense smaller quadtree nodes to form larger ones when filling regions. It achieves an $O(B)$ time complexity similar to the algorithms described above, where B is the number of boundary pixels. In practical comparisons, the proposed algorithm fills faster without an increase in space requirements.

2.0 Useful Properties of Location Codes

By inspecting the quaternary location code of an individual pixel, it is possible to determine the relationships between that pixel and the lower resolution cells which contain it (i.e., lie on the path between it and the root of the quadtree). These relationships are formalized in the following set of rules (the examples are for a resolution $r = 4$. See Figure 1).

Rule1. Given the number k of trailing quaternary zeros of a location code m, the size of the cell which m can represent is not larger than $(2^k \times 2^k)$, $k \geq 0$

e.g. $m = 2100_4$ can represent:
(2100, 4) of size 1 x 1 (pixel)
(2100, 3) of size 2 x 2
(2100, 2) of size 4 x 4

Rule 2. Given the number k of identical trailing quaternary digits of a location code m, the size of the cell for which m can form one of the vertices is not larger than $(2^k \times 2^k)$, $k \geq 1$

e.g. $m = 2111_4$ forms the SE vertex of:
(2110, 3) of size 2 x 2
(2100, 2) of size 4 x 4
(2000, 1) of size 8 x 8

The value of the trailing digit indicates which vertex of the cell m forms
0:SW, 1:SE, 2:NW, 3:NE

To find the location codes of the cells, substitute the k digits by 0 from right to left and use rule 1 to get the size of the cell.

Rule 3. Given the number k of trailing quaternary digits of a location code m where the set of trailing digits comprise two distinct quaternary digits the size of the cell for which m can form the edge or diagonal of is not larger than $(2^k \times 2^k)$, $k \geq 2$

e.g. $m = 2323_4$ forms part of the N edge of:
(2300, 2) of size 4 x 4
(2000, 1) of size 8 x 8
(0000, 0) of size 16 x 16

or $m = 3202_4$ forms part of the W edge of:
(3200, 2) of size 4 x 4
(3000, 1) of size 8 x 8

or $m = 1330_4$ forms part of the main diagonal of:
(1300, 2) of size 4 x 4
(1000, 1) of size 8 x 8

The two values of the trailing digits indicate which edge or diagonal of the cell m forms
(0,1):S, (0,2):W, (2,3):N, (1,3) E
(0,3) main diag. (1,2) cross diag.

To find the location codes of the cells, substitute the digits by 0 from right to left and use rule 1 to get the size of the cell.

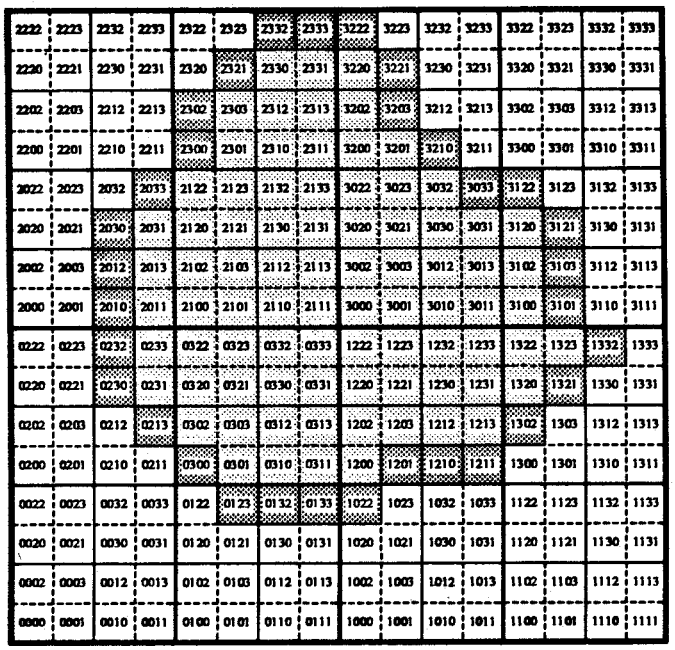

Figure 1 - LQT domain r=4 with filled region

To get a better understanding of the rules listed above, consider the LQT domain in Figure 1 ($r = 4$) as a meta-cell of size $2^4 \times 2^4$ within a larger domain of resolution r'. Setting $r' = 7$ for this example, the quaternary location codes for the pixels in the cell will have the form $dddnnnn_4$, where the digits ddd give the location of the cell within the encompassing domain, and the digits nnnn give the location of the pixels within the cell, and are identical to the quaternary location codes shown in Figure 1. Recall that the location codes are formed by bit interleaving the X and Y coordinates of the pixels: each quaternary digit of the location code corresponds to two bits, an X and a Y bit, with the X bit in the least significant position.

In the example, the SW vertex of the meta-cell has the local coordinate (0, 0) within the cell and must therefore have the location code $ddd0000_4$. Applying rule 1 to this location code, $k = 4$ and the size of cell represented by this code is $2^4 \times 2^4$, which is the size of the cell in Figure 1.

The SE, NW, and NE vertices of the cell have local coordinates (2^r-1, 0), (0, 2^r-1) and (2^r-1, 2^r-1)

respectively, which when interleaved, form the location codes $ddd1111_4$, $ddd2222_4$, and $ddd3333_4$. Applying rule 2 to these location codes, $k = 4$, and the size of cell having these locations as vertices is also $2^4 \times 2^4$.

Consider now a pixel along the edge of the meta-cell. Since either the X or Y coordinate along a particular edge is fixed, only one bit in each quaternary digit can change and therefore each digit can take on only one of two values. The allowed values of the digits along an edge will be fixed by the values of the digits of the quaternary location codes forming the vertices of that edge. The southern edge of the cell in Figure 1, for example, lies between $ddd0000_4$ and $ddd1111_4$, and the location codes of the pixels along the edge joining them have the form dddnnnn where the digit n can only be quaternary 0 or 1. A similar observation can be made about the location codes of the pixels lying along the diagonal between two vertices of a cell. In this case, the X and Y coordinates change at the same rate along the diagonal, and therefore both bits of any quaternary digit will change at the same time, also allowing only one of two values for the digits.

Applying rule 3 to the location code $ddd1010_4$, for instance, $k = 4$ and therefore the size of cell which it forms the edge of is $2^4 \times 2^4$ which agrees with Figure 1.

3.0 Filling by Insertion and Traversal

The proposed filling algorithm consists of two phases. In the first phase, the boundary pixels are inserted into a regular quadtree. As the pixels are inserted, the non-pixel nodes are marked either black or grey depending on the spatial and blocking relationship between the pixel being inserted and the node being traversed. After the insertion phase, the quadtree contains black and grey nodes. Uncoloured nodes are represented by null branches in the tree. The black nodes in the quadtree correspond to the largest LQT cells forming the boundary of the filled region (Figure 5).

The second phase of the algorithm traverses the quadtree created in the first. When a black node is encountered, its location and level are output to the region LQT. When a null-branch is encountered,

the colour of the cell it represents is determined. If the cell is found to be black, its location and level are also output to the region LQT. If a grey node is encountered its children are traversed.

The resulting LQT represents the filled region, and the list of location codes are in sorted order, a result of traversing the quadtree in preorder.

3.1 Insertion Phase

The data structure used to store the input boundary pixels is a simple regular quadtree. Figure 2(a) shows the structure for a single node. Since the quaternary location code of the boundary pixel describes the path through the regular quadtree from root to leaf, insertion is straightforward. The task consists of following the branch indicated by the next significant quaternary digit of the location code, and creating new nodes where necessary.

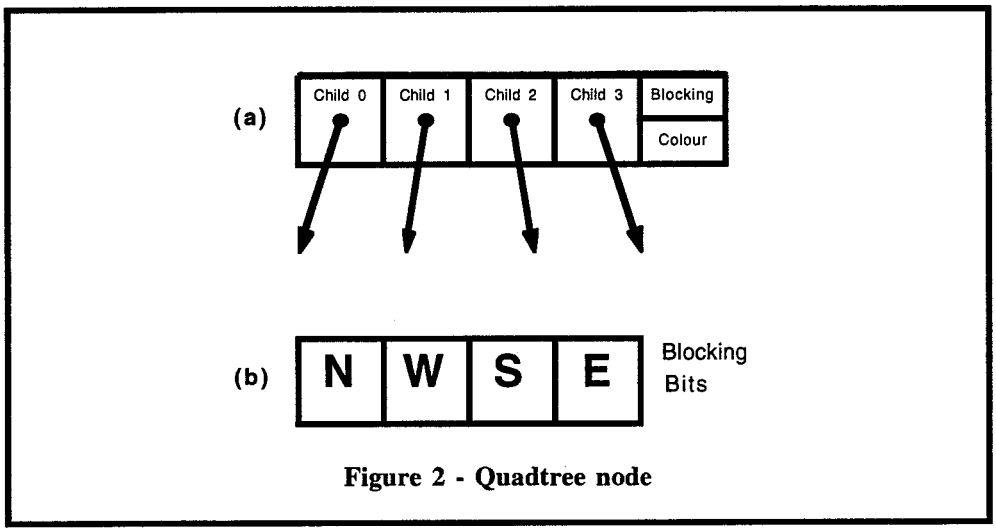

Figure 2 - Quadtree node

Using the rules given in Section 2, it is possible to determine before insertion the cells which contain the boundary pixel as an edge pixel or vertex. pixel. Such cells are represented by nodes which lie along the path between the root of the quadtree and the leaf representing the boundary pixel. The blocking code of the boundary pixel being inserted indicates which side(s) of the pixel is exterior to the region. This information must also be used to determine if the non-leaf nodes must be grey, or are potentially black.

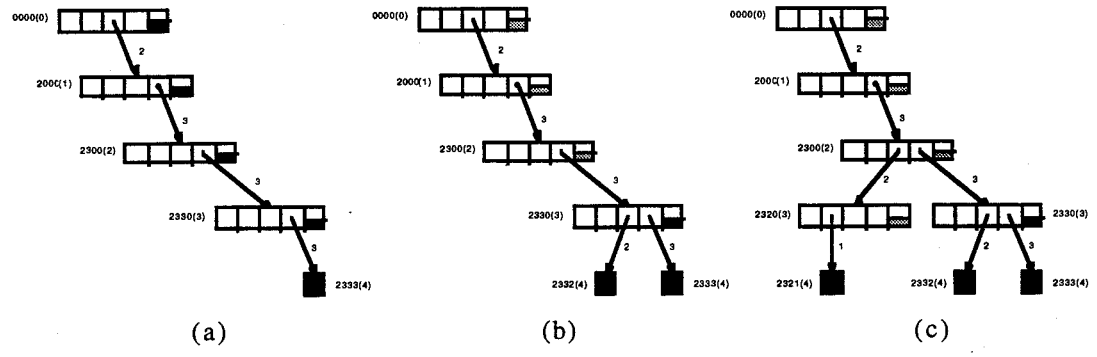

Figure 3 - Insertion sequence

When a quadtree node is created, it is initially marked black. As a node is traversed during the insertion process, it will be marked grey if the pixel being inserted does not fall on the edge or corner of the cell represented by the node. Similarly, if the pixel being inserted does lie on the edge or vertex of the cell being traversed, the blocking code of the pixel may exclude that cell from being black, in which case it is also marked grey. Referring to the example boundary in Figure 1, suppose that boundary pixels are inserted starting in the order 2333_4, 2332_4, 2321_4. Pixel 2333_4 is blocked only on its northern side, and it sits on the northern edge of cell (0000, 0) and on the NE vertex of cells (2000, 1), (2300, 2), (2330, 3). This means, without knowing any additional information about other boundary pixels, that all four cells could potentially be black.

Therefore, the nodes representing these cells are created and coloured black as the pixel is inserted (Figure 3(a)). From the location code of the second pixel, 2332_4, it can be determined that it sits on the northern edge of cells of level 0, 1, 2 and at the NW vertex of a cell of level 3. Also, it is blocked on the west, and therefore all of the cells that the pixel forms the northern edge of must be marked grey. The cell which it forms the vertex of remains marked black (Figure 3(b)). The third pixel, 2321_4, does not form the edge of a larger cell. It does form the SE vertex of a cell of level 3, but since it is blocked in the north and west directions, the level-3 cell must also be grey (Figure 3(c)). Figure 4 gives the complete pseudocode for the insertion process.

```
Quadtree Insert (in pixel location, in blocking code)

BEGIN
        FIND vertex and maximum vertex cell size GIVEN location code
        FIND edge and maximum edge cell size GIVEN location code

        DETERMINE if pixel is a valid vertex GIVEN blocking code and vertex
        DETERMINE if pixel is a valid edge GIVEN blocking code and edge

        level := 0
        node := root

        WHILE (level < maximum vertex cell level) DO
                IF (level < maximum edge cell level) OR (pixel is not valid edge) THEN
                        node colour := grey
                END IF
                node block code := node block code | pixel block code
                child := next significant quaternary digit of pixel location code
                IF (child node does not exist) THEN create black child node
                node := child node
        END WHILE

        WHILE (level < quadtree depth)
                node block code := node block code | pixel block code
                IF (pixel is not valid vertex) THEN node colour := grey
                child := next significant quaternary digit of pixel location code
                IF (child node does not exist) THEN create black child node
                node := child node
        END WHILE
END
```

Figure 4 - Insertion pseudocode

3.2 Traversal Phase

Once all of the boundary pixels of a region have been inserted into the quadtree, the resulting structure is traversed in preorder. Each time a black node is encountered, its location code and level are output as part of the LQT representing the filled region. Any children of a black node are ignored. If a grey node is encountered, then its children are visited. Note that the LQT location code of each cell can be generated from the current position in the quadtree traversal.

The linear quadtree generated by the traversal described above is the sorted list of all of the largest cells forming the boundary of the region. Figure 5 shows the result for the original example. Cells which are left uncoloured correspond to branches of the quadtree which were left null after the insertion phase. The problem remaining is to determine their colour.

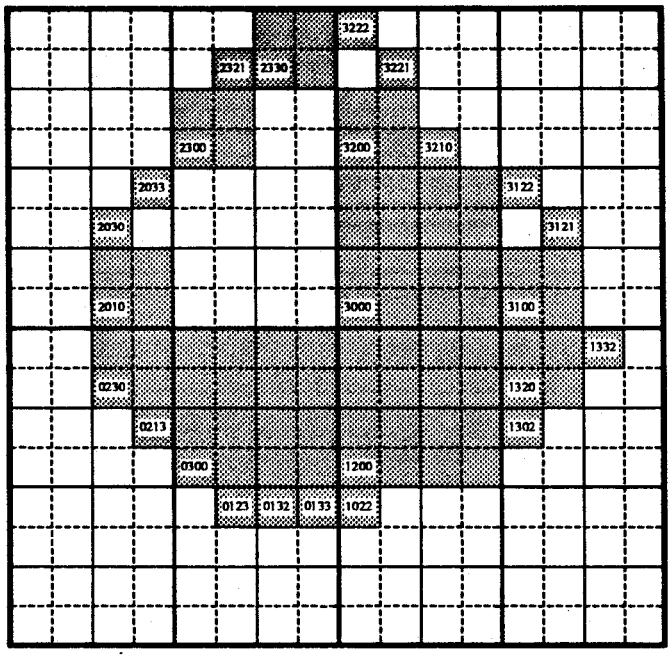

Figure 5 - Boundary cells

To facilitate cell colouring, a field is included in each quadtree node (Figure 2a) which contains the composite blocking code of all boundary pixels contained by that node. The blocking code consists of four bits, one for each direction (Figure 2b). When a bit in the blocking code is set, it indicates that the corresponding edge of the associated cell or pixel forms the perimeter of the region.

During the insertion phase, the blocking code of a pixel is bitwise ORed with the composite blocking codes of the nodes it traverses as it is inserted. Thus if a pixel contained by a node has a blocking bit set for a particular direction, the composite blocking code will also have that bit set. When insertion is complete, grey nodes may contain blocking codes which are not valid since the boundary may be concave within the cell. Such a case is illustrated in Figure 6 where the composite blocking code indicates that the cell is blocked on all sides whereas, the cell is only blocked on the north and west sides. Therefore, only the black nodes are considered to have valid blocking codes after insertion.

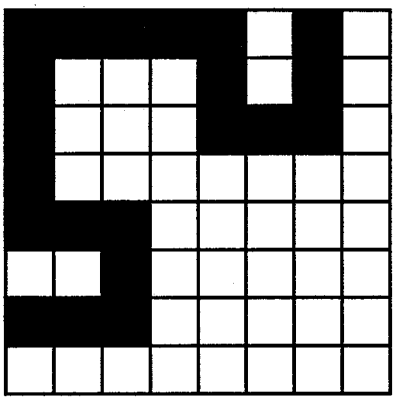

Figure 6 - Example of cell with invalid blocking code

The colour of an uncoloured cell (null pointer in the quadtree) can be determined from the blocking code of its black or grey siblings. If the cell lies to the outside of its black or grey sibling then its colour must be white, otherwise, it must be black, and its location code and level can be recorded in the output LQT. Since a grey sibling node does not have a valid composite blocking code, the subtree rooted at that node must be searched until a black node (having by definition a valid blocking code) is found. The search is carried out so that the node finally arrived at is as close as possible to the shared edge of the grey and uncoloured siblings. The blocking code of the black node found in the search is used to determine if the uncoloured node lies to the inside or outside of the grey sibling.

If more than one uncoloured node belong to the same parent node, it is only necessary to find the colour of one of these siblings. The colours of the other siblings can be inferred from the colour assigned to the first sibling. When two uncoloured siblings are edge neighbours of one another, they cannot be separated by the region boundary and therefore must be assigned the same colour. When neighbours are located across a vertex, however, it is possible for the two uncoloured siblings to be separated by the boundary and they will be assigned the opposite colours. Cells (2200, 2) and (2100, 2) in Figure 1 are an example. The condition can be detected since there will be two grey or black siblings which are neighbours of each other across a vertex. Figure 7 gives the pseudocode for the recursive traversal process.

```
Quadtree Traverse (in Quadtree, in level, in current location code)

BEGIN
        IF (level = quadtree depth) THEN RETURN

        FOR (child = 0 TO 3) DO
                DERIVE child location code GIVEN current location code and child number
                IF (child node exists) THEN
                        IF (child node is grey) THEN
                                Quadtree Traverse (child node, level+1, child location code)
                        ELSE
                                OUTPUT child location code and level to LQT
                        END IF
                ELSE (child node must be coloured)
                        IF (sibling has been coloured) THEN
                                IF (coloured sibling is edge neighbour) AND (is black) THEN
                                        OUTPUT child location code and level to LQT
                                ELSE IF (coloured sibling is vertex neighbour) AND (is white) AND
                                        (other two siblings exist) THEN
                                        OUTPUT child location code and level to LQT
                                END IF
                        ELSE
                                DETERMINE colour of child GIVEN blocking code of edge neighbour
                                IF (colour is black) THEN OUTPUT child location code and level to LQT
                        END IF
                END IF
        END FOR
END
```

Figure 7 - Traversal pseudocode

3.3 Saving Time and Space

The traversal phase of the filling algorithm described in the previous section checks for the colour of non-null nodes. Since pixel nodes have no children, and can only take the colours black or white, they do not need to be explicitly stored. It is sufficient that a non-null pointer (dummy pointer) exists in the parent of the pixel node to indicate the presence of a black pixel. The space that would be required for a quadtree node for each boundary pixel can thereby be saved. For a complete quadtree, the amount of storage required by the pixels would be 4^r units, while the space required by all other nodes in the tree would be $\sum_0^{r-1} 4^k = \frac{4^r-1}{3}$. In this best case there is a 75% saving in space by eliminating the pixel nodes, while in more realistic cases, as determined experimentally for randomly generated boundaries, the space saving is on average better than 50%.

Therefore, the quadtree traversal algorithm is modified to output an LQT cell not only when a black

node is encountered, but also when a dummy pointer is encountered at the parent of pixel level in the quadtree.

The algorithm for colouring an uncoloured cell searches the subtree rooted at the grey sibling, until a node with a valid blocking code is found. In the previous section, cells with valid blocking codes after the insertion phase were defined only as those that were coloured black. Since the search for a black cell can no longer reach the pixel level, given the new storage scheme, the definition of *cell with valid blocking code* must be redefined to include all cells which are parents of boundary pixels. In order to guarantee that the parent of a pixel node does indeed have valid composite blocking codes, there can be no conflicting blocking codes in the pixels inserted through them. Blocking codes conflict when two neighbouring pixels with the same parent are blocked along the edge that they have in common. Since, from the definition of the valid input data, a boundary cannot double back on itself in a concave bend (see section 4.5 and Figure 8(a),(b)) all conflicting blocking codes are avoided in the parent of pixel cells.

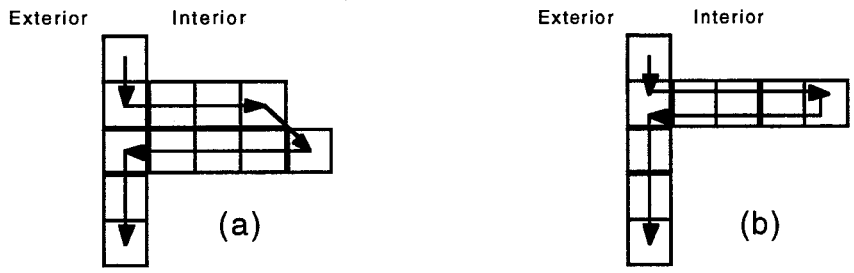

Figure 8 - Boundary pixels which must be pruned

3.4 Finding Missing Pixels

In the new quadtree storage scheme, boundary pixel nodes are represented by dummy pointers at the parent of pixel level. Since the pixel nodes are represented differently than other nodes in the quadtree, the colouring algorithm described earlier will not work for uncoloured pixels. The colour must therefore be determined as the boundary pixels are inserted, rather than during the quadtree

traversal phase.

The colour of uncoloured pixels will only need to be determined if the parent is coloured grey, since otherwise they would not be visited in the traversal. An uncoloured internal pixel will occur in a grey parent only if two boundary pixels contained by that parent are vertex neighbours of each other. In this case, the uncoloured pixels will also be vertex neighbours of one another, and their colours will be opposite. Pixel set 3120_4, 3123_4 of Figure 1 is an example. To determine which of the two is black, the blocking code of the second boundary pixel is examined as it is inserted. The uncoloured pixel on the side of the newly inserted pixel which is unblocked is coloured black by setting the corresponding pointer in the parent to the dummy value.

3.5 Input Data

The input data comprises a set of pixels which define the boundary of a given region on a quadtree domain. Each pixel is stored as an LQT location code with an associated blocking code. The order of the pixels may be arbitrary, but the entire group must form one or more closed loops which do not intersect each other.

Data can be generated in many different ways but in all cases the input data must meet the following criteria.

1 - The boundary is not self intersecting unless the blocking codes are set such that the outsides of loops which are formed have the same parity.

2 - A boundary does not double back on itself in a concave corner (Figure 8). In 8(a), internal pixels are blocked along an edge shared with another internal pixel. In 8(b), internal pixels are blocked on both the north and south edges. In both cases, since the pixels are interior to the filled region, they are not real boundary pixels, and their blocking codes should be zero, or the pixels should be eliminated.

3 - Pixels may be repeated, but may not have contradicting blocking codes.

The second criterion implies that a certain amount of preprocessing is necessary to ensure the validity of the boundary. The preprocessor must prune all of those boundary pixels which are completely

contained by the filled region, that is, all internal appendages (Figure 8a,b). An equivalent step is to set their blocking codes to zero. Preprocessing is necessary for instance when the boundary pixels are generated from concave polygons where internal angles exceed a certain threshold causing internal appendages to be formed.

Note that the requirement to prune the boundary is common to all other algorithms using blocking codes. Some of the algorithms, such as Gargantini's, are robust enough to handle the cases shown in Figure 8.

4.0 Runtime Analysis

The execution time of the boundary filling depends on the number of nodes visited in each phase of the algorithm.

During the boundary pixel insertion phase, each pixel which is inserted must visit one node per resolution level of the quadtree. At each level, it must be determined whether the appropriate child node exists. If it does not, then a new node must be created. The cost of node creation can be distributed among all of the boundary pixels inserted into the tree. In general, the number of nodes created per pixel insert becomes progressively less as more pixels are created.

For a quadtree of resolution r, a pixel must visit r nodes as it is inserted into the tree. Insertion occurs once per boundary pixel, so given n boundary pixels, the insertion process is $O(rn)$ or simply O(n), since r is constant for a given application.

During the quadtree traversal phase, the quadtree is traversed in preorder. If a black node is encountered, the cell represented by that node is output, and all of the children of that node are ignored. Since black nodes correspond to the largest cells adjoining the boundary of the region (Figure 5), there are at worst $O(n)$ black nodes, and the cost of visiting them is also $O(n)$. Encountering a null node means that its colour must be determined. This is achieved by searching down the levels of the subtree rooted at a non-null sibling such that the first black node is found which

is as close as possible to the shared boundary of the non-null and uncoloured siblings. The search will visit at most r nodes. Since the number of uncoloured nodes is proportional to the number of black nodes, the cost of finding colours will also be $O(n)$.

5.0 Verification

The proposed filling algorithm was verified primarily with a random boundary generating tool. The resulting filled regions were displayed on a raster monitor, where any holes were immediately obvious. As a double check, the output LQTs were compared with those generated by the competing algorithms for the same boundaries and were found to be identical.

6.0 Algorithm Assessment and Comparison[1]

In order to test the merits of the filling algorithm described in this thesis, it was compared with two recently published algorithms that operate on the same type of boundary descriptions to derive filled regions expressed as linear quadtrees.

The first of these algorithms is that of Atkinson, Gargantini and Walsh [8] which is a refinement of an algorithm published earlier by Gargantini and Atkinson [9]. It uses blocking codes to find missing cells, starting at the pixel level of the quadtree, working toward the root, propagating blocking codes from children to parent. The effect is to grow the region inward from the boundary to the centre. This algorithm requires as data structures a linked list to store the input boundary pixels and blocking codes and a linked list to store the output LQT which stores location codes and levels.

The algorithm presented by Mark and Abel [10] operates on a sorted list of boundary pixels to derive runs of black and white pixels. The black runs are easily converted into LQT cells. In order to keep the execution time of the overall algorithm linear, a bucket sort was used to sort the input data. The time to sort the input and the time to convert from runs to LQT cells were included in the measured time of execution. The algorithm requires two arrays, one to store the input boundary pixels, and the

[1] The algorithms were implemented in ANSII C, and compiled with the gcc compiler using the optimizer flag (-O). The tests were run on a Sun IPX.

other to store the runs of black pixels (defined by start and stop pixel location codes).

Figure 9 gives a graphical comparison of the three algorithms in terms of execution time vs. boundary length for 100 randomly generated boundaries. Clearly, all three algorithms are linear. The proposed algorithm is, on average, 40% faster than the Gargantini algorithm, which has the second best time performance.

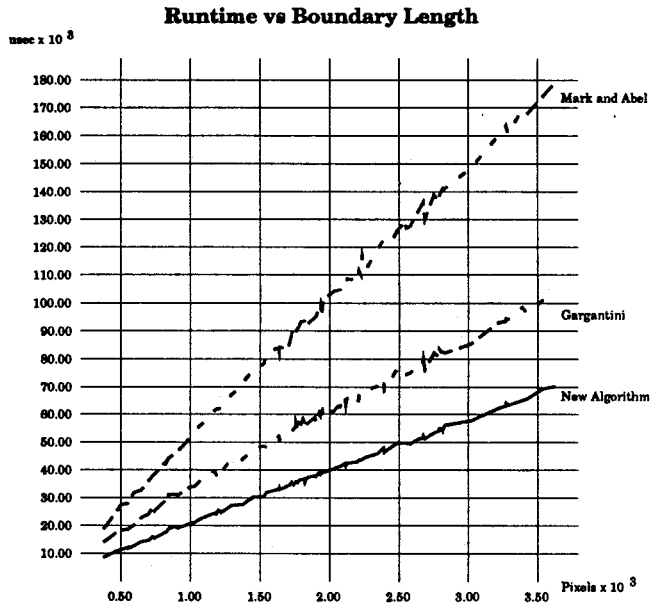

Figure 9 - Runtime comparison of algorithms

More detailed timing measurements of the two competing algorithms show that sorting the input data comprises the largest single time expense. The rearranging of data during a sort requires many read and write accesses to memory, which slows the overall performance. By avoiding a sort, the proposed algorithm achieves its superior runtime performance.

Figure 10 shows the total memory in bytes allocated to each algorithm vs. the boundary length. In this case, the proposed algorithm uses slightly less memory than the Gargantini algorithm, but slightly more than the Mark and Abel algorithm. The chief difference here is that the Mark and Abel algorithm does not require storage for pointers since the data are stored contiguously in arrays.

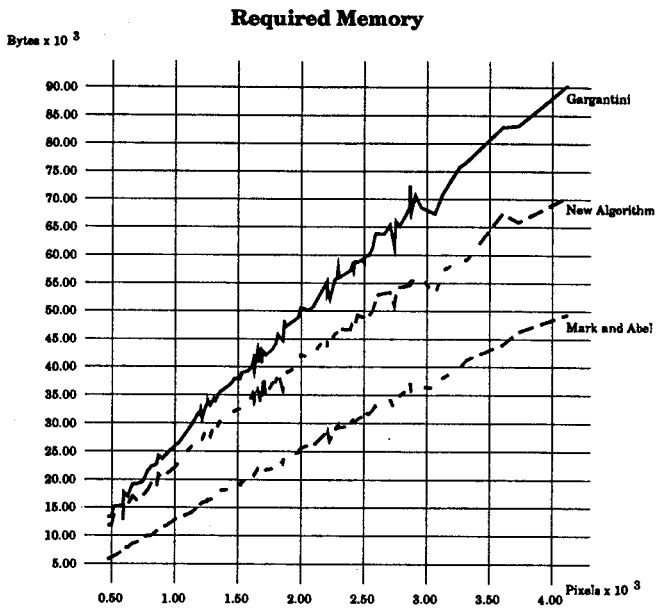

Figure 10 - Memory requirements of algorithms

The relationship between memory requirements for the first two algorithms can change slightly, depending on the compiler and system used. The results for the Gargantini algorithm would be slightly better if the data could be stored on other than four byte boundaries, since the input blocking code only requires a single byte.

7.0 Extension to Octrees

The algorithm described in the previous sections can be easily extended to work in the octree domain. The changes to the algorithm are all related to the addition of a third dimension. Instead of filling regions defined by boundary pixels, as for the quadtree case, the task becomes one of filling a volume defined by a set of surface voxels.

Several analogies can be formed between the quadtree and octree domains. A cell in the quadtree domain is represented by a node within the quadtree. It corresponds to a square subdivision of the entire two-dimensional domain. Similarly, a cell in the octree domain is also represented as a node within the octree, and corresponds to a cubic subdivision of the three-dimensional domain. The octree node has eight rather than four children, each child corresponding to one of the eight equal

subdivisions of the cubic volume represented by the node. Octree cells can also be represented by a location code and level within the octree, just as in the case for quadtrees. For octrees, the location code is also formed by interleaving the coordinates of the domain, in this case X, Y, and Z. When expressed as an octal number, each digit of the location code corresponds to the next branch of the tree, when traversing the tree from root to leaf, serving the same role as quaternary location codes in linear quadtrees.

The three rules presented in section 2 relating pixels to the quadtree cells which contain them can stand almost unchanged as the relationship between cells and voxels in octrees. References to quaternary digits and numbers must simply be substituted with references to octal digits and numbers and references to the size of cells must include the third dimension (i.e. $2^k \times 2^k \times 2^k$ rather than $2^k \times 2^k$). A fourth rule can be formulated, which relates the voxel to the octree cells which it forms the face of. Changing the context to the octree domain then, the rules are as follows. Note that in the examples here, the X and Y axis still fall along the east west and north south directions, and that the Z axis falls along the up-down (U/D) axis.

Rule1. Given the number k of trailing octal zeros of a location code m, the size of the cell which m can represent is not larger than $(2^k \times 2^k \times 2^k)$, $k \geq 0$

e.g. $m = 2100_8$ $r = 4$ can represent:
(2100, 4) of size 1 x 1 x 1 (voxel)
(2100, 3) of size 2 x 2 x 2
(2100, 2) of size 4 x 4 x 4

Rule 2. Given the number k of identical trailing octal digits of a location code m, the size of the cell for which m can form one of the vertices is not larger than $(2^k \times 2^k \times 2^k)$, $k \geq 1$

e.g. $m = 2111_8$ $r = 4$ forms the DSE vertex of:
(2110, 3) of size 2 x 2 x 2
(2100, 2) of size 4 x 4 x 4
(2000, 1) of size 8 x 8 x 8

The value of the trailing digit indicates which vertex of the cell m forms
0:DSW, 1:DSE, 2:DNW, 3:DNE
4:USW, 5:USE, 6:UNW, 7:UNE

To find the location codes of the cells, substitute the k digits by 0 from right to left and use rule 1 to get the size of the cell.

Rule 3. Given the number k of trailing octal digits of a location code m where the set of trailing digits comprise two distinct octal digits the size of cell for which m can form the edge, face diagonal, or internal diagonal of is not larger than $(2^k \times 2^k \times 2^k)$, $k \geq 2$

e.g. $m = 2323_8$ r=4 forms part of the DN edge of:
(2300, 2) of size 4 x 4 x 4
(2000, 1) of size 8 x 8 x 8
(0000, 0) of size 16 x 16 x 16

or $m = 7646_8$ r=4 forms part of the UW edge of:
(7600, 2) of size 4 x 4 x 4
(7000, 1) of size 8 x 8 x 8

or $m = 1550_8$ r=4 forms part of the main diagonal of the S face of:
(1500, 2) of size 4 x 4 x 4
(1000, 1) of size 8 x 8 x 8

The two values of the trailing digits indicate which edge or diagonal of the cell m forms
(0,1):DS, (0,2):DW, (2,3):DN, (1,3) DE
(4,5):US, (4,6):UW, (6,7):UN, (5,7) UE
(0,4):SW, (1,5):SE, (2,6):NW, (3,7) NE
(0,3) main diag. (1,2) cross diag D face
(4,7) main diag. (5,6) cross diag U face
(0,5) main diag. (1,4) cross diag S face
(2,7) main diag. (3,6) cross diag N face
(0,6) main diag. (2,4) cross diag W face
(1,7) main diag. (3,5) cross diag E face
(0,7), (1,6), (3,4) internal diagonals

To find the location codes of the cells, substitute the digits by 0 from right to left and use rule 1 to get the size of the cell.

Rule 4. Given the number k of trailing octal digits of a location code m where the set of trailing digits comprise three or four distinct octal digits the size of cell for which m can form the face of is not larger than $(2^k \times 2^k \times 2^k)$, $k \geq 3$

e.g. $m = 571023_8$ r=6 forms part of the D face of:
(571000, 3) of size 8 x 8 x 8
(570000, 2) of size 16 x 16 x 16

or $m = 572046_8$ r=4 forms part of the W face of:
(572000, 3) of size 8 x 8 x 8
(570000, 2) of size 16 x 16 x 16

The values of the trailing digits indicate which face of the cell m forms
(0,1,2,3): D, (4,5,6,7): U
(0,1,4,5): N, (2,3,6,7): S
(0,2,4,6): W, (1,3,5,7): E

To find the location codes of the cells, substitute the digits by 0 from right to left and use rule 1 to get the size of the cell.

```
Octree Insert (in voxel location, in blocking code)

BEGIN
        FIND vertex and maximum vertex cell size GIVEN location code
        FIND edge and maximum edge cell size GIVEN location code
        FIND face and maximum face cell size GIVEN location code

        DETERMINE if pixel is a valid vertex GIVEN blocking code and vertex
        DETERMINE if pixel is a valid edge GIVEN blocking code and edge
        DETERMINE if pixel is a valid face GIVEN blocking code and face

        level := 0
        node := root

        WHILE (level < maximum edge cell level) DO
                IF (level < maximum face cell level) OR (pixel is not valid face) THEN
                        node colour := grey
                END IF
                node block code := node block code | pixel block code
                child := next significant quaternary digit of pixel location code
                IF (child node does not exist) THEN create black child node
                node := child node
        END WHILE

        WHILE (level < maximum vertex cell level) DO
                node block code := node block code | pixel block code
                IF  (pixel is not valid edge)  node colour := grey
                child := next significant quaternary digit of pixel location code
                IF (child node does not exist) THEN create black child node
                node := child node
        END WHILE

        WHILE (level < quadtree depth)
                node block code := node block code | pixel block code
                IF (pixel is not valid vertex) THEN node colour := grey
                child := next significant quaternary digit of pixel location code
                IF (child node does not exist) THEN create black child node
                node := child node
        END WHILE
END
```

Figure 11 - Octree insertion pseudocode

The algorithm for insertion described in section 3.1 must be modified to check if the surface voxel being inserted into the octree forms part of the face of some of the cells which contain it. It must then examine the blocking code of the voxel, to see if the cells which the pixel forms the face of may remain black, or mus be marked grey. The rest of the algorithm remains essentially unchanged and is shown in Figure 11. After the insertion phase, the largest cells forming the surface of the object, are all marked black in the octree.

The traversal algorithm described in section 3.2 also remains essentially unchanged. Colouring uncoloured cells becomes more complex in that colouring based on the juxtaposition of uncoloured siblings is not straightforward. Given that an uncoloured cell has been assigned a colour after searching a black or grey sibling for a valid blocking code, other uncoloured neighbours may be assigned the same colour provided that they are not separated by the surface of the object. This is the case for instance when the uncoloured siblings are neighbours across a face. Determining whether two uncoloured siblings are or are not separated by the surface when they are not neighbours across a face is done by examining the black or grey siblings, to see if they contain surface voxels which separate them. Alternatively, uncoloured cells which are not face neighbours with one another can be assigned colours using the search technique used to assign colour to the first uncoloured sibling. Figure 12 shows the pseudocode for the modified traversal algorithm.

```
Octree Traverse (in Octree, in level, in current location code)

BEGIN
        IF (level = octree depth) THEN RETURN

        FOR (child = 0 TO 7) DO
                DERIVE child location code GIVEN current location code and child number
                IF (child node exists) THEN
                        IF (child node is grey) THEN
                                Octree Traverse (child node, level+1, child location code)
                        ELSE
                                OUTPUT child location code and level to LOT
                        END IF
                ELSE (child node must be assigned colour)
                        IF (siblings have been assigned colour) THEN
                                IF (coloured sibling is face neighbour) AND (is black) THEN
                                        OUTPUT child location code and level to LOT
                                ELSE
                                        DETERMINE colour of child GIVEN block code of face neighbour
                                        IF (colour is black) THEN OUTPUT child location and level to LOT
                                END IF
                        ELSE
                                DETERMINE colour of child GIVEN blocking code of face neighbour
                                IF (colour is black) THEN OUTPUT child location code and level to LOT
                        END IF
                END IF
        END FOR
END
```

Figure 12 - Traversal pseudocode

8.0 Conclusions

The algorithm presented in this thesis marries the advantages of linear and regular quadtrees to derive an algorithm for filling boundaries defined by closed loops of pixels whose exterior is consistently defined by associated blocking codes. The normal space disadvantages of quadtrees is partially overcome by not explicitly storing the boundary pixels in the quadtree. White and non-boundary cells are also not stored. By traversing the quadtree formed from inserting the boundary pixels in preorder, sorting of the data is avoided, a step which has been shown to slow other similar filling algorithms.

The thesis introduces the concept of using the boundary pixel location codes to establish the relationship between those pixels, and the quadtree cells which contain them (i.e., whether they lie on their edges or vertices). This information is used in conjunction with the pixel blocking codes to colour the quadtree cells black or grey as the boundary pixels are inserted. The resulting quadtree has all of the boundary cells marked black and condensing cells in the filling process is thereby avoided.

The resulting filling algorithm is a two-phase algorithm which first inserts the boundary pixels into a regular quadtree and later traverses the tree, searching for black nodes, and colouring uncoloured siblings of the black nodes.

Other filling algorithms operating on boundaries in the LQT domain exist. The most recent of these [11, 12] also require the input boundary pixels to be sorted. Since the sorting process requires the majority of the runtime in both the Gargantini and the Mark and Abel algorithms, the time performance of the newer algorithms is not expected to be better than the proposed algorithm.

Like the Gargantini algorithm, the proposed algorithm is easily extensible to the octree domain. The main problem, unrelated to filling algorithms, lies in generating the surface voxels and blocking codes from the vertices of a polyhedron.

References

[1] Morton, G. M., "A computer-oriented geodetic data base and a new technique in file sequencing", Technical Report, IBM Ltd., Ottawa, Canada, March 1966.

[2] G. Schrack, and I. Gargantini, "Mirroring and rotating images in linear quadtree form with few machine instructions", Image Vision and Computing, vol. 11, no. 2, pp. 112-118, March 1993.

[3] G. Schrack, "Finding neighbours of equal size in linear quadtrees and octrees in constant time", CVGIP: Image Understanding, vol. 55, no.3, pp. 221-230, May 1992.

[4] I. Gargantini, "Translation, rotation and superposition of linear quadtrees", Int. J. Man-Machine Studies, vol. 8, no.3, pp. 253-263, March 1983.

[5] I. Gargantini, "An effective way to represent quadtrees", Communications of ACM, vol 25, pp. 905-910, 1982.

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

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

[8] H. H. Atkinson, I. Gargantini, and T.R. Walsh, "Filling by quadrants and octants", Computer Vision, Graphics, and Image Processing, vol. 33, pp. 138-135, 1986.

[9] I. Gargantini, and H. H. Atkinson, "Linear quadtrees: A blocking technique for contour filling", Pattern Recognition, vol. 17, no. 3, pp. 258-293, 1984.

[10] D. Mark, and D. Abel, "Linear quadtrees from vector representations of polygons", IEEE Trans. on Pattern Analysis and Machine Intelligence, vol PAMI-7, no. 3, pp. 344-349, May 1985.

[11] M. R. Lattanzi and C.A. Shaffer, "An optimal boundary to quadtree conversion algorithm", Computer Vision, Graphics, and Image Processing, vol. 53, no. 3, pp. 303-312, 1991.

[12] Shi-Nine Yang and Tsong-Wuu Lin, "A new linear octree construction by filling algorithms", IEEE Tenth Annual International Phoenix Conference on Computers and Communications, pp 740-746, 1991.

[13] M. L. V. Pitteway, "Algorithm for drawing ellipses or hyperbolae with a digital plotter", Computer J., vol. 10, no. 3, pp. 282-289, November 1967.

[14] J. Foley, A. van Dam, S. Feiner and J. Hughes, *Computer Graphics: Principals and Practice, Second Edition,* Addison-Wesley, Reading, MA, 1990.

Appendix A

Some Useful Results

A1 Discrete Rotations and Reflections in the LQT Domain

A useful result to emerge from research into the patterns and symmetries of LQT location codes is a new method of performing a limited set of transformations on objects defined as LQTs. This set of transformations entails: (a) the identity transformation, (b) mirroring about the X axis, (c) mirroring about the Y axis, (d) rotation of 180 degrees (about the centre of the LQT domain), (e) mirroring about the main diagonal, (f) mirroring about the cross diagonal, (g) rotation of 90 degrees (CCW), and (h) rotation of 270 degrees (CCW). The expressions derived below for these transformations take fewer machine instructions than previously developed algorithms performing the same functions [2], and the methodology used provides a new view of LQT location codes.

The common feature of the transformation set is that the transformations can be expressed as a mapping of one LQT domain onto another. In these transformations, the location codes of each pixel in one domain are changed into the location codes of the new domain, in other words, there is a 1:1 mapping between pixels. Equally important to note is that the changes to each quaternary digit of the location code will be the same for a particular transformation. For example, referring to the LQT domain in Figure 1, a mirroring about the *X*-axis will change 0's to 2's, 2's to 0's, 1's to 3's and 3's to 1's. There is no interaction between the digits inside the location codes. Thus, to derive an expression for the transformations, one needs only consider the changes required for a single quaternary digit inside the location code.

With this in mind, consider the simplified LQT domain of r=1, and all of its transformations (Figure A1). To find an expression for each case, a truth-table is constructed relating the transformed to the untransformed digit (Table A1) where the binary representation is used for the location code digit. From the truth-table, the boolean expression for each bit can be derived for a particular transformation, as shown in Table A2. Here d_x^T indicates bit x of the transformed digit, and d_x' indicates the complement of bit x of the untransformed digit.

$d_1\ d_0$	a	b	c	d	e	f	g	h
0 0	0 0	1 0	0 1	1 1	0 0	1 1	1 0	0 1
0 1	0 1	1 1	0 0	1 0	1 0	0 1	0 0	1 1
1 0	1 0	0 0	1 1	0 1	0 1	1 0	1 1	0 0
1 1	1 1	0 1	1 0	0 0	1 1	0 0	0 1	1 0

Table A1 - Truth Table for Bits of Quaternary Digits

	a	b	c	d	e	f	g	h
d_1^T =	d_1	d_1'	d_1	d_1'	d_0	d_0'	d_0'	d_0
d_0^T =	d_0	d_0	d_0'	d_0'	d_1	d_1'	d_1	d_1'

Table A2 - Boolean Expressions for Bits of Quaternary Digits

It is clear from the boolean equations that all transformations can be expressed in terms of swapping and complementing the original bits. In order to implement these functions, individual bits of the location code must be selectively complemented. This becomes possible by taking advantage of the following useful properties of the boolean exclusive OR function:

$$A \oplus 0 = A$$
$$A \oplus 1 = A'$$

Thus, for the single digit example for mirroring about the X-axis, we can simplify the expression given in Table A2b to:

$$d^T = d \wedge 2$$

where the operator ^ is the C notation for bitwise exclusive OR. Bit swapping can be achieved by selecting the most and least significant bits of the quaternary digit using bitwise AND, shifting the bits one position right and left respectively, and bitwise ORing the results. Thus the bit swap required for the transformation shown in Table A2e used to mirror about the main diagonal can be expressed as:

$$d^T = ((d\&1)<<1) \mid ((d\&2)>>1)$$

where the operators& and | are the C notations for bitwise AND and OR, and **>>x** and **<<x** are the notations for arithmetic shift right and left respectively.

In order to extend the results derived above to location codes of arbitrary length, the bit masks t_x, t_y and t_n are introduced (as defined in [2]), where:

$$t_x = \sum_0^{r-1} 2^{2i} \quad = (00...0101..01),$$
$$t_y = (t_x << 1) \quad = (00...1010..10), \text{ and}$$
$$t_n = (t_x \mid t_y) \quad = (00...1111..11)$$

Now, for example, complementing the most significant bits of each quaternary digit in a location code can be expressed as $(loc \wedge t_y)$, selecting the least significant bits of the quaternary digits as $(loc \,\&\, t_x)$ and complementing all bits of the location code as $(loc \wedge t_n)$.

Using these bit masks, the expressions for all of the transformations of Table A2 for location codes of arbitrary length can now be written as follows:

(a) $loc^T = loc$
(b) $loc^T = loc \wedge t_y$
(c) $loc^T = loc \wedge t_x$
(d) $loc^T = loc \wedge t_n$
(e) $loc^T = ((loc \,\&\, t_x) << 1) \mid ((loc \,\&\, t_y) >> 1)$
(f) $loc^T = (((loc \,\&\, t_x) << 1) \mid ((loc \,\&\, t_y) >> 1)) \wedge t_n$
(g) $loc^T = (((loc \,\&\, t_x) << 1) \mid ((loc \,\&\, t_y) >> 1)) \wedge t_x$
(h) $loc^T = (((loc \,\&\, t_x) << 1) \mid ((loc \,\&\, t_y) >> 1)) \wedge t_y$

From the above expressions, it becomes clear that some transformations can be expressed as simple combinations of other transformations. A rotation of 270 degrees (h) for instance, is simply a mirroring about the main diagonal (e) followed by a mirroring about the *X*-axis (b). Mirroring about the cross diagonal (f) consists likewise of a mirroring about the main diagonal (e), followed by a rotation of 180 degrees (d).

In order to transform a cell of lower than pixel resolution (i.e. non-pixel leaf node), the location code of the cell is transformed using one of the above formulae as for a pixel. The least significant (*r-level*) quaternary digits are then set to zero, where level is the depth of the cell in the quadtree. In Figure 1, for example, if the cell (3030, 3) is to be mirrored about the *Y* axis, transform (b) is applied to 3030 giving 2121. The least significant digit is then set to zero, the result being the location code for the transformed cell, (2120, 3).

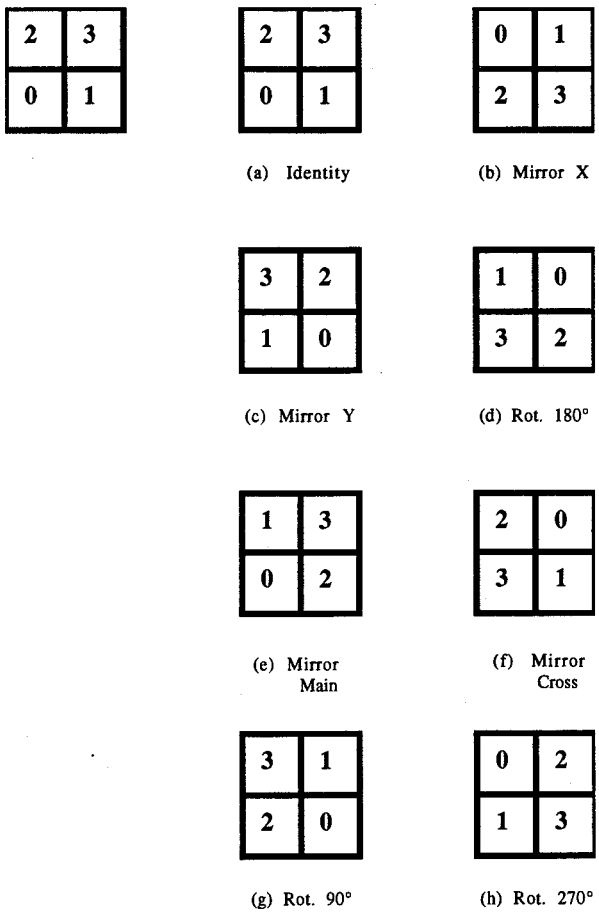

Figure A1 - Eight discrete transformations of the LQT domain r = 1

Since each quadrant and subquadrant of the LQT domain is also an LQT domain of lower resolution, the formulae for the transformations can be applied within a quadrant. To do this we simply redefine the mask t_x such that $t_x = \sum_0^{r'-1} 2^{2i}$, where r' = *r-level*. t_y and t_n are still defined the same way in terms of t_x. Applying the redefined formulae to location codes will leave the first *r-level* most significant digits unchanged thus allowing the transformations to occur at cell level. Figure A2 shows an LQT object which is transformed at cell-*level* = 1.

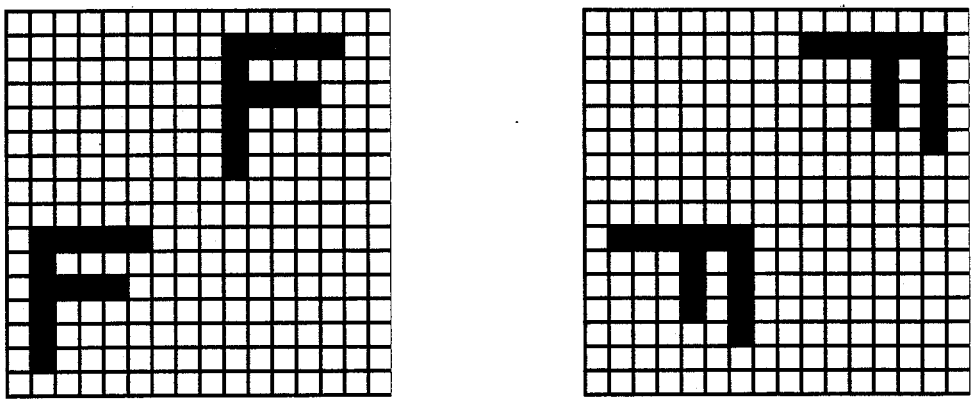

Figure A2 - Rotating cells by 270°

A2 Converting from Chain Code to Boundary Pixels and Blocking Codes

A chain code or Freeman chain consists of a series of numbers, each representing edges of unit length and discrete direction, which is used to describe a single closed loop on a raster. Using the analogy of a driver navigating a car through city streets, the chain code can be thought of as a list of directional instructions to be followed by the driver such that a new instruction is taken from the top of the list at each intersection encountered (e.g. go left, go right, go straight, go back). The area enclosed by the loop is defined as being either to the left or the right of the boundary (depending on convention) as it is traversed in the order defined by the chain code. (In mathematics, convention normally defines the interior as lying to the left.)

The chain code is used in one of two complementary ways. The first type of chain code is used to represent a series of raster edges which separate pixels interior to the region from pixels exterior to it. There are four possible directions (E N S W) each of which is assigned a code number (e.g. E:0, N:1, W:2, S:3). Figure A3(a) shows an area defined by this method. The location of a pixel on the raster must be given as the starting point for the loop.

The second type of chain code, often called a Freeman chain, is such that each element of the chain represents the direction from the current pixel to the next pixel along the boundary. Since, on a raster,

the next pixel is necessarily a neighbour of the current, there are eight possible directions, which for the purpose of the example in figure A3(b) are numbered E:0, NE: 1, N:2, NW:3, W:4, SW:5, S:6, SE:7. The Freeman chain also requires the specification of a pixel which defines the start of the loop.

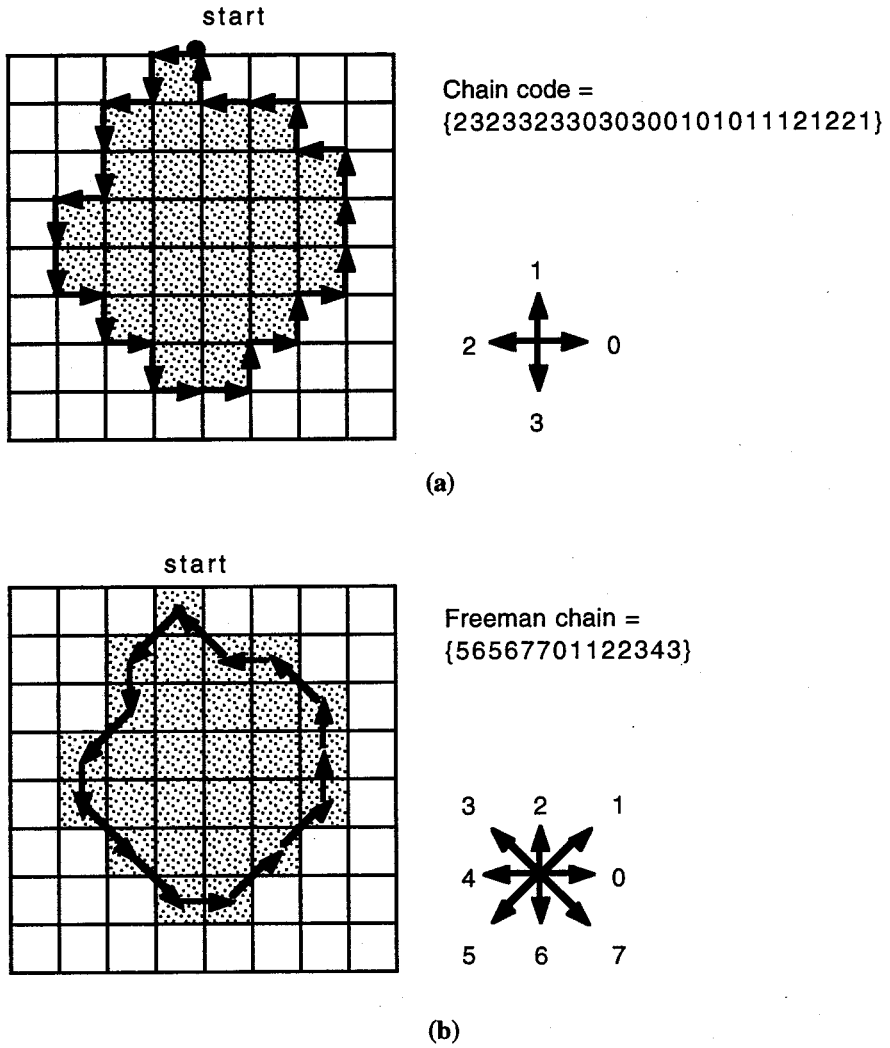

Figure A3 - Regions defined by chain codes

The first type of chain code was not used to define boundary pixels in the research done for this thesis. A method for converting chain codes of this type to boundary pixels and blocking codes is presented by Samet in [6, 7]. Note that the method for finding neighbours used in Samet's algorithm can be substituted by Schrack's method [3] when working with linear rather than regular quadtrees.

Conversion of chain codes of the second type into boundary pixels is trivial since each element of the chain code indicates which pixel relative to the current one forms the next pixel along the boundary. By successively applying a neighbour finding algorithm for each new boundary pixel generated, all of the boundary pixels can be found. Schrack's neighbour finding algorithm is an obvious choice, since it works in constant time, and can be implemented as a register function (i.e. it is very fast).

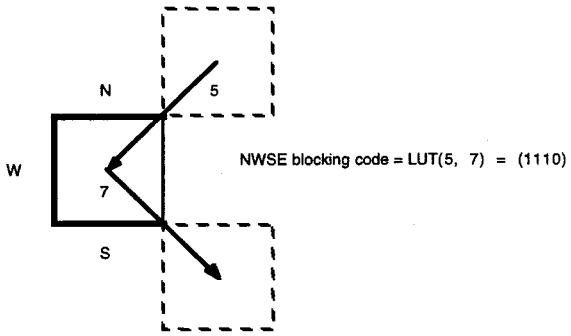

Figure A4 - Freeman chain to blocking code conversion

Finding the blocking code for each boundary pixel is accomplished using a two dimensional look-up table (LUT). The indices of the LUT are the values of the current and previous chain code values for each new pixel found. These values indicate which neighbours of the current pixel form the boundary of the region from which can be inferred which sides of that pixel border on the exterior of the region. Table A3 shows the LUT and Figure A4 gives an example of it's use.

	0	1	2	3	4	5	6	7
0	0010	0010	0000	0000	1110	1110	0110	0110
1	0011	0011	0001	0001	0000	0000	0111	0111
2	0011	0011	0001	0001	0000	0000	0111	0111
3	1011	1011	1001	1001	1000	1000	0000	0000
4	1011	1011	1001	1001	1000	1000	0000	0000
5	0000	0000	1101	1101	1100	1100	0100	0100
6	0000	0000	1101	1101	1100	1100	0100	0100
7	0010	0010	0000	0000	1110	1110	0110	0110

Table A3 Look-up table to convert freeman codes to blocking codes (NWSE) row indices = previous code / column indices = current code

A3 Scan Converting Lines and Circles in the Quadtree Domain

In computer graphics, the most common representation of objects is by polygons defined by the Cartesian coordinates of their vertices. Finding the pixels which comprise the edges is achieved by scan converting the lines defined by successive vertex pairs of the polygon. Such a conversion is necessary, for instance, when the proposed filling algorithm is used to fill a region defined by a polygon, since the required input for the algorithm is the set of all boundary pixels of the region.

Many scan conversion algorithms for lines on the standard Cartesian raster are known. One in particular is well suited to be adapted to the LQT domain such that the location codes of pixels along the line being scan-converted are derived directly without the need to encode the coordinates of the pixels. This algorithm is called the *Midpoint Line Algorithm* and was first published by Pitteway in 1967 [13]. A thorough description of this algorithm is presented in [14], a summary of which is given below, along with the details for implementing it in the LQT domain.

For the discussion which ensues, assume that lines have a slope between 0 and 1 and that other slopes can be handled by appropriate reflections about the principal axis. The lower left endpoint is called (x_0, y_0) and the upper right endpoint (x_1, y_1).

Figure A5 shows a line in the process of scan conversion. Here, (x_p, y_p) is the pixel P which has just been selected as part of the scan converted line, and the next pixel must be either the eastern (E) or the northeastern (NE) neighbour. Let Q be the intersection of the line being scan converted with the line $x = x_p + 1$. If the midpoint M between the eastern and northeastern neighbours lies above Q then the eastern neighbour must be the next pixel in the scan converted line , otherwise the northeastern neighbour will be the next pixel.

By writing the equation of the line in implicit form, and substituting the coordinates for the next midpoint, a decision variable d can be defined which will determine which neighbour (E or NE) will form the next scan converted pixel. Thus if the function for the line being converted is

$$F(x, y) = ax + by + c = 0, \text{ then}$$
$$d = F(M) = a(x_p+1) + b(y_p+\tfrac{1}{2}) + c$$

If $d > 0$ then the next scan converted pixel is the NE neighbour otherwise if $d<0$ the E neighbour will be chosen as the next pixel. Note that if the line has the explicit form $y = \frac{dx}{dy}x + B$, $a = dy$, $b=-dx$ and $c = B \cdot dx$.

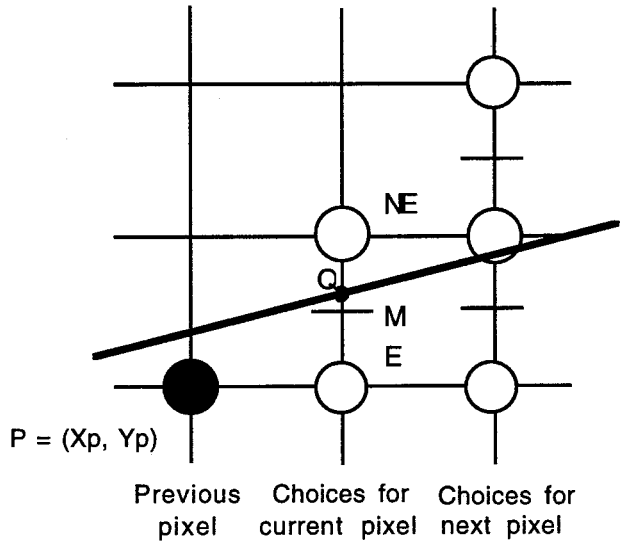

Figure A5 - Scan conversion of a line

It is possible to express the decision variable for the current pixel d_{new} in terms of the previous decision variable d_{old}. For example, if an eastern neighbour was just chosen as the current scan converted pixel then

$$d_{new} = a(x_p+2) + b(y_p+\tfrac{1}{2}) + c, \text{ and}$$
$$d_{old} = a(x_p+1) + b(y_p+\tfrac{1}{2}) + c$$

and $d_{new} = d_{old} + a$. Similarly, if a NE neighbour was chosen,

$$d_{new} = a(x_p+2) + b(y_p+\tfrac{3}{2}) + c$$

and $d_{new} = d_{old} + a.+ b$. Thus only the original decision variable d_{start} needs to be explicitly evaluated.

$$
\begin{aligned}
d_{start} &= F(x_0+1, y_0+\tfrac{1}{2}) \\
&= a(x_0+1) + b(y_0+\tfrac{1}{2}) + c \\
&= F(x_0, y_0) + a + \tfrac{1}{2}b \\
&= a + \tfrac{1}{2}b, \quad \text{since } F(x_0, y_0) = 0
\end{aligned}
$$

In order keep to integer arithmetic, we multiply d_{start} by 2 when evaluating it, since only the sign of the decision variable is important. As a consequence, we must multiply the increments to the decision variable by 2 also. Figure A6 gives the complete algorithm for scan converting a line of slope between 0 and 1 in the LQT domain. Note that Schrack's neighbour finding algorithm [3] can be used here since it is fast, and requires no encoding or decoding. If the vertices of the line are given as location code, then no initial encode is required, however, a decode would be required in order to calculate dx and dy.

```
Line to LQT pixels (in vertex0, in vertex1, out LQT_pixels)

BEGIN
        dx      = vertex1.x - vertex0.x
        dy      = vertex1.y- vertex0.y
        d       = 2 * dy - dx
        IncrE   = 2 * dy
        IncrNE  = 2 * (dy - dx)

        location = encode(vertex0)
        OUTPUT (location)

        WHILE  (location ≠ encode(vertex1)) DO
                IF (d ≤ 0) THEN
                        d = d + IncrE
                        location = neighbour(location, E)
                ELSE
                        d = d + IncrNE
                        location = neighbour(location, NE)
                END IF
                OUTPUT(location)
        END WHILE
END
```

Figure A6 Scan conversion pseudocode for line in LQT domain

Finding the blocking code for the pixels generated by the scan conversion is trivial. Assuming that the interior of the region lies to the left of the line, the pixel will be blocked on the south if it was chosen as an eastern neighbour, and will be blocked on the east and south if it was chosen as a north-eastern neighbour.

A similar set of decision variables as those described above can be derived for the scan conversion of circles. The resulting scan conversion algorithm uses only integer arithmetic and avoids multiplications while generating pixels. A discussion of this technique can be found in [14] by the interested reader. Important to note here is that by use of an LQT neighbour finding algorithm, the scan conversion algorithm can be modified to generate the location codes of the circle's boundary pixels directly (without encoding individual pixels). Figure A7(a) gives the modified version of the algorithm. It takes advantage of the symmetry of the circle by generating the pixels for only one 45 degree segment. The other symmetrical pixels are derived by using the mirroring and rotation techniques described in Appendix A1. Figure A7(b) gives the algorithm for generating 8 symmetrical pixels given one input pixel. Note that the circle generated is centred on the origin. To shift the centre of the circle, the LQT pixels can be translated as they are generated by using the dilated integer arithmetic presented in [3].

A4 Converting from Runs to LQT Cells

It is sometimes convenient to express objects in the LQT domain as a number of runs of black or white pixels in Morton sequence order. A run of pixels is defined by a start and an end pixel, whose locations are given as LQT location codes. Such object descriptions are used as the output from the Mark and Abel filling algorithm [10]. To convert a run of black pixels into LQT cells for comparison with output from the proposed filling algorithm, an method was developed which operates on the location codes of the start and end pixels of the runs.

```
Circle to LQT pixels (in radius)

BEGIN
        d       = 1 - radius
        delE    = 3
        delSE   = -2 * radius + 5

        location = encode(0, radius)
        Write Circle Pixels(location)

        WHILE  (location & t_x < ((location & t_y) <<1 )) DO
                IF (d ≤ 0) THEN
                        d = d + delE
                        delE = delE + 2
                        delSE = del SE + 2
                        location = neighbour(location, E)
                ELSE
                        d = d + delSE
                        delE = delE + 2
                        delSE = del SE + 4
                        location = neighbour(location, SE)
                END IF
                Write Circle Pixels(location)
        END WHILE
END
```

(a)

```
Write Circle Pixels (in location)

BEGIN
        OUTPUT(location)
        OUTPUT(Mirror X (location))
        OUTPUT(Mirror Y (location))
        OUTPUT(Rotate 180 (location))
        OUTPUT(Mirror Main Diagonal (location))
        OUTPUT(Mirror Cross Diagonal (location))
        OUTPUT(Rotate 90 (location))
        OUTPUT(Rotate 270 (location))
END
```

(b)

Figure A7 Scan conversion pseudocode for circle in LQT domain

To understand the conversion algorithm, it is useful to know how to convert an LQT cell into a Morton sequence run of pixels. Recall that an LQT cell is described by the location code of its SW corner (in the convention used in this thesis) and by its level within the quadtree measured from the root of the tree. The start pixel of the run is thus trivially the location code of the LQT cell. The last pixel of the

LQT cell when following the Morton sequence is the NE corner which, according to rule 2 of section 2 will have a quaternary location code formed by substituting trailing zeros of the cell location code by trailing threes such that the number of zeros substituted is equal to the resolution of the quadtree domain minus the level of the LQT cell (*r-level*). For example the NE corner pixel of (21000, 3) is 21033_4, where r=5 in this case.

```
Run To LQT (in blackrun, out LQTcells)

BEGIN
        newstart := 0
        start    := run.start
        i        := 0
        WHILE (newstart ≤ run.end) DO
                Find Largest Cell (start, run.end, cellsize, newstart)
                LQTcells[i].location_code := start
                LQTcells[i].level         := cellsize
                start                     := newstart
                increment i
        END WHILE
END


Find Largest Cell (in start, in end, out cellsize, out newstart)

BEGIN
        i          := 0
        NE corner := start
        WHILE (NE corner < stop) DO
                IF (next least significant quaternary digit of NE corner is zero) THEN
                        substitute quaternary 0 with the digit 3
                        increment i
                ELSE
                        EXIT WHILE
                END IF
        END WHILE
        newstart := NE corner + 1
        cellsize  := r-i
END
```

Figure A8 Run To LQT conversion algorithm

Given a run of black pixels in Morton sequence order, the task of converting to LQT cells consists of finding the largest LQT cell having the same location code as the starting pixel of the run, but whose NE corner pixel has a location code which is not greater than the end pixel of the run. Having found

this cell, the black run is shortened such that the new start pixel of the run has a location code one greater than that of the NE corner of the cell just found. The process is repeated until the new run has zero length, and all of the LQT cells contained in the run have been found. Figure A8 gives the pseudocode for the algorithm.