REPRESENTING IMAGES USING THE QUADTREE DATA STRUCTURE (HEBREW CONSONANTS AND VOWELS)

By

ISRAEL SHUVAL

Bachelor of Science

Oklahoma City University

Oklahoma City, Oklahoma

1986

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July, 1989

THESIS
1989
S562r
Cop 2.

Oklahoma State Univ. Librar

REPRESENTING IMAGES USING THE QUADTREE DATA STRUCTURE (HEBREW CONSONANTS AND VOWELS)

Thesis approved:

M. Samadzadeh-H.
Thesis Advisor

R. E. Haskell

J P Chandler

Norman N. Durham
Dean of the Graduate College

1224432

PREFACE

This thesis is a discussion and application of the quadtree data structure for storing images and in particular, representing the Hebrew consonants and vowels. The study includes a design and implementation using the Amdek 286A microcomputer system (Amdek Corporation, 1901 Zanker Road, San Jose, CA 95112 USA).

I would like to express my sincere appreciation and gratitude to my major advisor Dr. M. H. Samadzadeh, for his guidance, motivation, encouragement, and invaluable assistance. I am also thankful to Drs. G. E. Hedrick and J. P. Chandler for serving on my graduate committee.

I am also grateful to Dr. D. D. Fisher for his constant faith and guidance during my studies.

I am grateful to Mr. and Mrs. Locke for their moral encouragement and direction.

Finally, my wife, Alexandria, my parents, Mr. and Mrs. Shvili, deserve my deepest appreciations for their love, understanding, and sacrifices throughout my studies.

TABLE OF CONTENTS

Chapter page

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

The use of hierarchical data structures is becoming increasingly important in the areas of computer graphics, image processing, computational geometry, geographic information systems, and robotics.

Hierarchical data structures are being used because of their efficient representation, improved execution times and ease of implementation. They are useful particularly for performing set operations. According to Samet[32], hierarchical data structures are currently used for point data, regions, curves, surfaces, and volumes. One example of a hierarchical data structure is a quadtree. Samet [32] defined the term quadtree as follows:

> The term quadtree is used to describe a class of hierarchical data structures whose common property is that they are based on the principle of recursive decomposition of space. They can be differentiated on the following bases:
>
> 1. the type of data that they are used to represent;
> 2. the principle guiding the decomposition;
> 3. the resolution (variant or not).

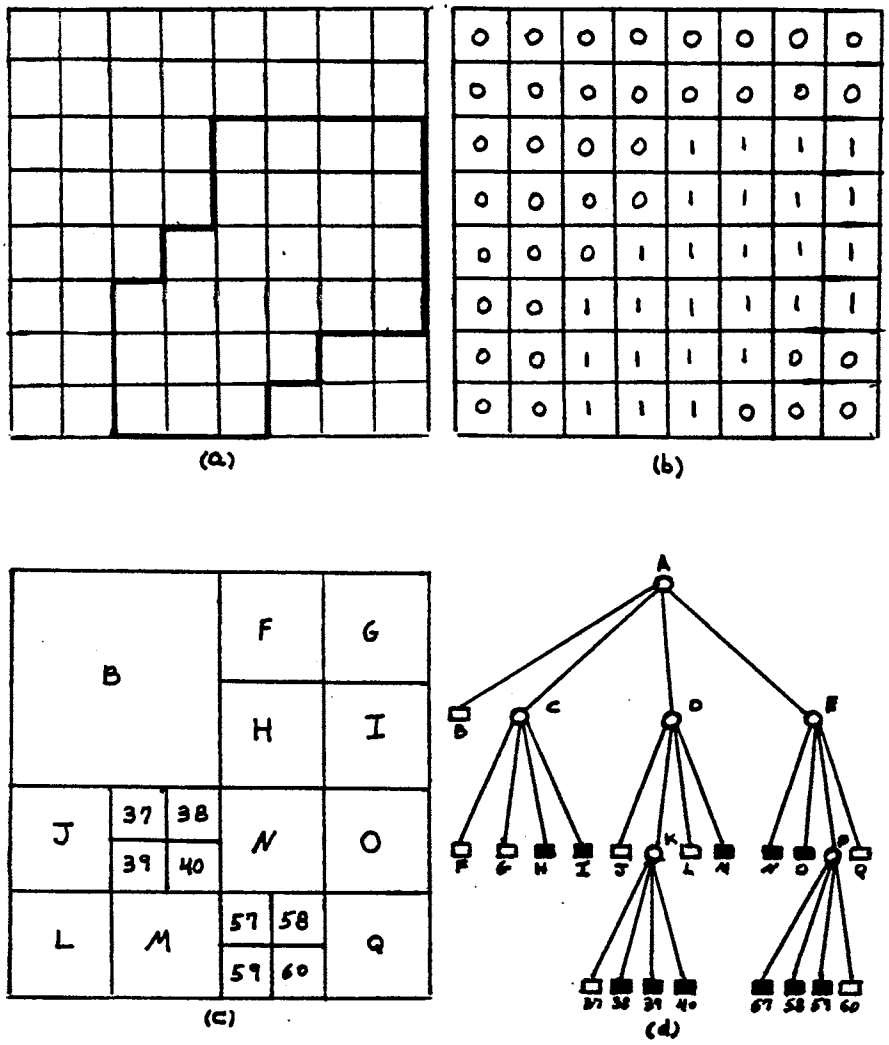
The region quadtree is the most studied quadtree approach for region representation. It is based on the successive subdivision of the image array into four equal-size quadrants.

The following illustrative example is from [32]. Consider the region shown in Figure 1a. The region is represented by the 2^3 by 2^3 binary array in Figure 1b. The 1's correspond to pixels that are in the region and the 0's correspond to the pixels that are outside the region.

If the region does not cover the entire array, it is subdivided into quadrants, subquadrants, etc. until blocks (possibly single pixels) that consist entirely of 0's are obtained; that is, each block is entirely contained in the region or entirely disjoint from it.

The resulting blocks for the array of Figure 1b are shown in Figure 1c. The process described above is represented by a tree of degree 4 (i.e., each nonleaf node has four sons). The root node corresponds to the entire array. Each son of a node represents a quadrant (labeled in order NW, NE, SW, SE) of the region represented by that node. The leaf nodes of the tree correspond to those blocks for which no further subdivision is necessary (i.e., blocks consisting entirely of 1's or entirely of 0's).

A leaf node is considered a BLACK node if its corresponding block is entirely inside of the represented region, WHITE otherwise. All nonleaf nodes are said to be



(Samet, H., "The quadtree and related hierarchical data structures." ACM Comput. Surveys, vol. 16, no. 2, Jun. 1984, pp. 190-235.)

Figure 1. (a) A region (b) Binary array (c) Block decomposition of the region (d) Quadtree representation of the blocks in (c).

GRAY. The quadtree representation for Figure 1c is shown in Figure 1d.

Objectives

The goal of this thesis is to use the quadtree data structure to represent the Hebrew consonants and vowels. In addition, a software will be developed to familiarize the beginning student of the Hebrew language. The programs were written in Turbo Pascal and were implemented on the Amdek 286A microcomputer system (Amdek Corporation, 1901 Zanker Road, San Jose, CA 95112 USA).

Thesis Organization

Chapter II presents the concept of region quadtree and alternative ways to represent quadtrees. Operations performed on quadtree are presented in Chapter III. Chapter IV illustrates the design and the implementation of thesis's goals described above. A summary and conclusions are included in Chapter V.

CHAPTER II

REGION DATA

A region can be represented in two major approaches:

1. those that specify the boundaries of a region
2. those that organize the interior of a region

The region quadtree, commonly referred to as a quadtree, is a hierarchical data structure that is characterized as being a collection of maximal blocks that partition a given region, for example, the run length code representation, where the blocks size is restricted to 1 by m rectangles. Another general representation treats the region as a union of maximal square blocks (or blocks of any other shapes) that may possibly overlap. In this representation, the blocks are usually specified by their centers and radii. This type of representation is called medial axis transformation (MAT) [32]. Samet [32] gives the following definition for a quadtree:

> The region quadtree is a variant on the maximal block representation. It requires that the blocks be disjoint and have standard sizes (i.e., sides of lengths that are powers of two) and standard locations. The motivation for its development was a desire to obtain a systematic way to represent homogeneous parts of

> an image. Thus, in order to transform the data into a region quadtree, a criterion must be chosen for deciding that an image is homogeneous (i.e., uniform). One such criterion is that the standard deviation of its GRAY level is below a given threshold t. By using this criterion the image array is successively subdivided into quadrants, subquadrants, etc. until homogeneous blocks are obtained. This process leads to a regular decomposition[32].

The blocks of the quadtree do not necessarily correspond to maximal homogeneous regions in the image. Most likely there exist unions of the blocks that are still homogeneous. To obtain a partition of the image into maximal homogeneous regions, merging of adjacent blocks (or unions of blocks) should be performed as long as the resulting region remains homogeneous.

Another method is to use a decomposition method that is not regular (i.e., rectangles of arbitrary size rather than squares). This alternative method may require less space, but it requires a search to determine the optimal partition points. The homogeneity criterion on which the subdivision process is done depends on the type of the region data.

From now on, the region quadtree (termed a quadtree in the rest of this paper) is assumed to have a domain of 2^n by 2^n binary images with 1 or BLACK corresponding to foreground and 0 or WHITE corresponding to background.

Alternative Ways to Represent Quadtrees

As is shown in Chapter I the most natural way to represent a quadtree is to use a tree structure. In this case each node is represented as a record with four pointers to the records corresponding to its sons. If the node is a leaf node, it will have four pointers to the empty record. Sometimes, in order to facilitate certain operations (e.g., merge) that require a reference of a node to its father, an additional pointer is included from a node to its father. This greatly eases the implementation of algorithms that perform basic image processing operations.

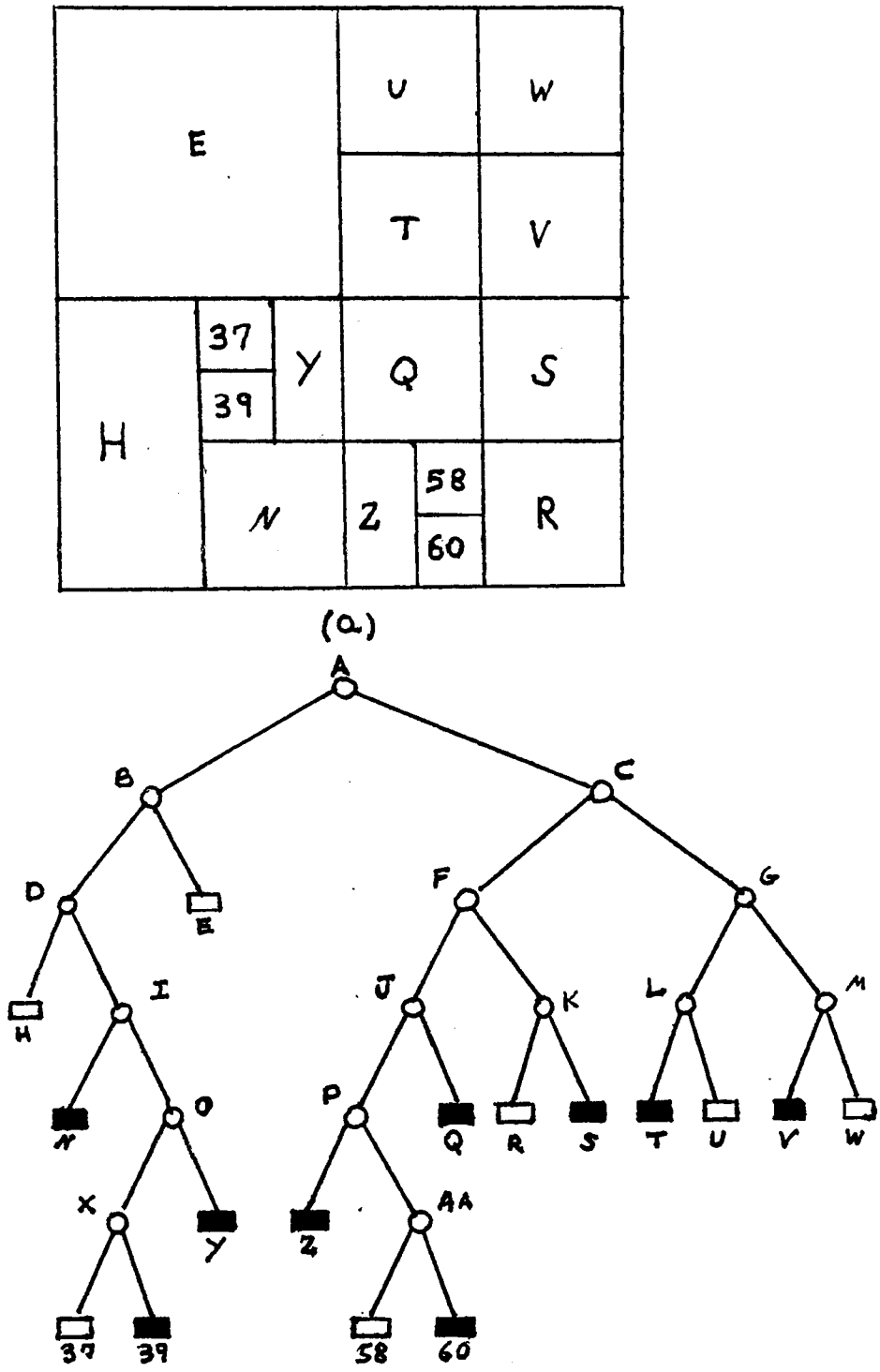
The problem with the tree representation of a quadtree is that it has a considerable amount of overhead associated with it. For example, a (B + W - 1)/3 additional nodes are necessary for the internal, i.e., GRAY, nodes (B ,W for the number of BLACK and WHITE, respectively). Moreover, each node requires additional space for the pointers to its sons. This is a problem when dealing with large images that cannot fit into core memory.

In order to solve the memory problem described above, there has been a considerable amount of interest in pointerless quadtree representations. These representations can be grouped into two categories [32]:

1. the image is treated as a collection of leaf node;
2. the image is represented in the form of a traversal of the nodes of its quadtree.

Bintree

The bintree structure reduces the number of pointers in each node. The space is always subdivided into two equal-sized parts alternating between the x and the y axes. Thus each node requires space only for pointers to its two sons instead of four sons (in the case of a quadtree). In addition, its use generally leads to fewer leaf nodes. Figure 2 is an example of a region and its corresponding bintree.



(Samet, H., "The quadtree and related hierarchical data structures." ACM Comput. Surveys, vol. 16, no. 2, Jun. 1984, pp. 190-235.)

Figure 2. (a) Block decomposition
(b) Bintree representation of the blocks in (a)

Linear Quadtree

In his paper [7], Garantini introduced a new structure, called "linear quadtree", with the following characteristics:

1. only BLACK nodes are stored;
2. the encoding used for each node incorporates adjacency properties in the four principal directions, namely NW, NE, SE, and SW;
3. the node representation implicitly encodes the path from the root to the node.

The main advantages of linear quadtrees, with respect to quadtrees, are:

1. space and time complexity depend only on the number of BLACK nodes (Garantini claimed a saving of 66 percent);
2. pointers are eliminated.

Encoding Black Pixels. Garantini adopted the following conventions [7]:

1. the NW quadrant is encoded with 0, the NE with 1, The SW with 2, and the SE with 3. Each BLACK pixel is then encoded in a weighted quaternary code, i.e., with digits 0, 1, 2, 3 in base 4, where each successive digit represents the

quadrant subdivision from which it originates. Thus, the digit of weight 4^{n-h}, $1 < h < n$, identifies the quadrant to which the pixel belongs at the hth subdivision (Figure 3).

2. The pair of integers (I,J), with I,J=0,1,.....,2^{n-1} identifies the position of a pixel in the $2^n X 2^n$ array. The encoding procedure consists of mapping the pair (I,J) into an integer K in base 4, which expresses the successive quadrants to which the (I,J) pixel belongs. For example, if n=3 and (I,J)=(6,5), K will be 321. This means that the pixel (6,5) belongs to the SE quadrant in the first subdivision, to the SW quadrant in the second, and to the NE in the third (final) subdivision (see Figure 3)

To find K, we first write I and J as

$$I = C_{n-1}2^{n-1} + C_{n-2}2^{n-2} + \ldots\ldots\ldots + C_0$$

$$J = D_{n-1}2^{n-1} + D_{n-2}2^{n-2} + \ldots\ldots\ldots + D_0$$

where C_i, D_i is either 0 or 1. Thus C_i and D_i indicate which quadrant (I,J) belongs to at each level of subdivision. Once the first (largest) quadrant is identified, we partition it into four quadrants and repeat the procedure for (n-2) and so on, until the last quadrant consists of only four elements.

In the previous example

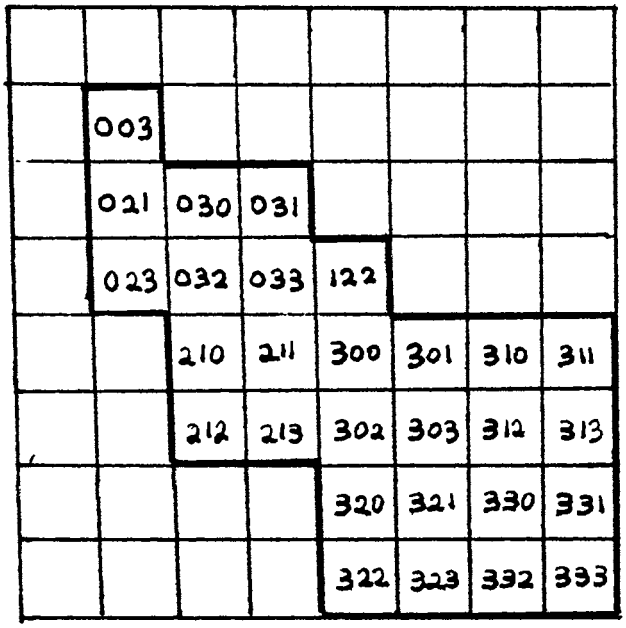
$$I = 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

$$J = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

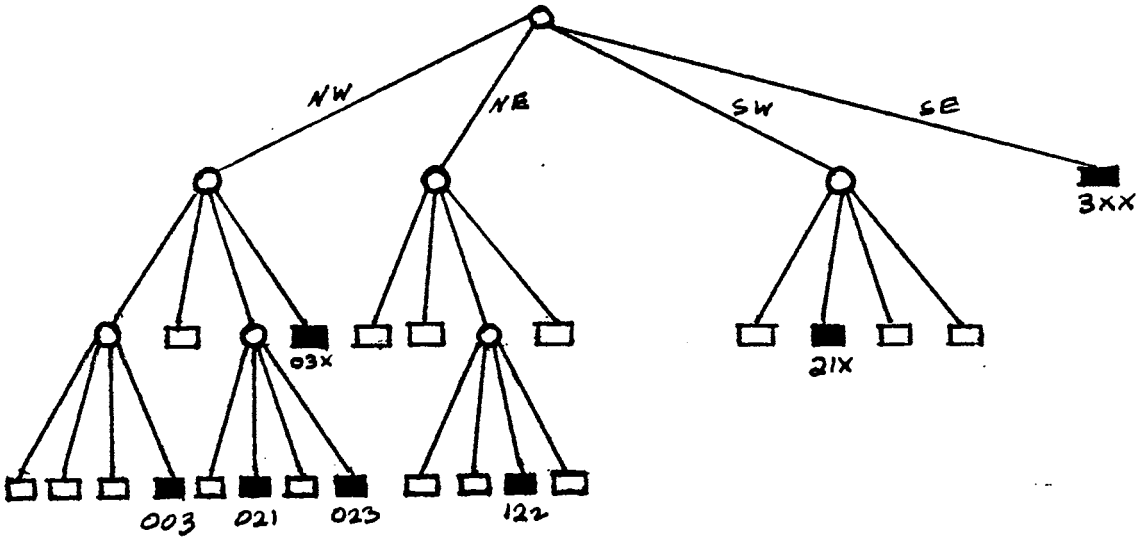
and thus, k=321.

After encoding all the BLACK pixels into their corresponding quadrant codes, we sort them and store them into an array. Then condensation is applied as follow: if four pixels have the same representation except for the last digit, we eliminate them from the list and replace them with a code of (n-1) quaternary digits followed by some kind of marker, here denoted by X.

For instance, if pixels 310, 311, 312, and 313 are all in the array, they are replaced by 31X. Similarly, if 30X, 31X, 32X, 33X are present, we replace them by 3XX and so forth. After condensation is complete, we obtain an array (containing only BLACK pixels or a covering thereof) which is still sorted if we suitably encode marker X with an integer greater than 3. This sorted array is referred to as the linear quadtree. The region shown in Figure 3b, for example, is encoded by the sequence, 003 021 023 03X 122 21X 3XX. Note that the linear quadtree corresponds to the postorder traversal of BLACK nodes of a quadtree, i.e., Figure 3b.



(a)



(b)

(Garantini, I., "An effective way to represent quadtrees." Commun. ACM, vol. 25, no. 12, Nov. 12, Dec. 1982, pp. 905- 910.)

Figure 3. (a) Quadrants Labeling and Generating Quaternary Codes
(b) Quadtree for region in (a)

DF-expression

Kawaguchi [13] introduced another pointerless representation of a quadtree. This representation is in the form of a preorder traversal (i.e., depth first) of the nodes of the quadtree. The result is a string consisting of the symbols "(", "B", and "W" corresponding to GRAY, BLACK, and WHITE nodes, respectively. For example, the image of Figure 4. has

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

as its DF-expression (assuming that sons are traversed in the order NW, NE, SW, SE). The original image can be reconstructed from the DF-expression by observing that the degree of each nonterminal (i.e., GRAY) node is always 4.

Kawaguchi [14] shows how a number of basic image processing operations can be performed on an image represented by a DF-expression. In particular, he demonstrates centroid computation, rotation, scaling, shifting, and set operations.

Forest Quadtree

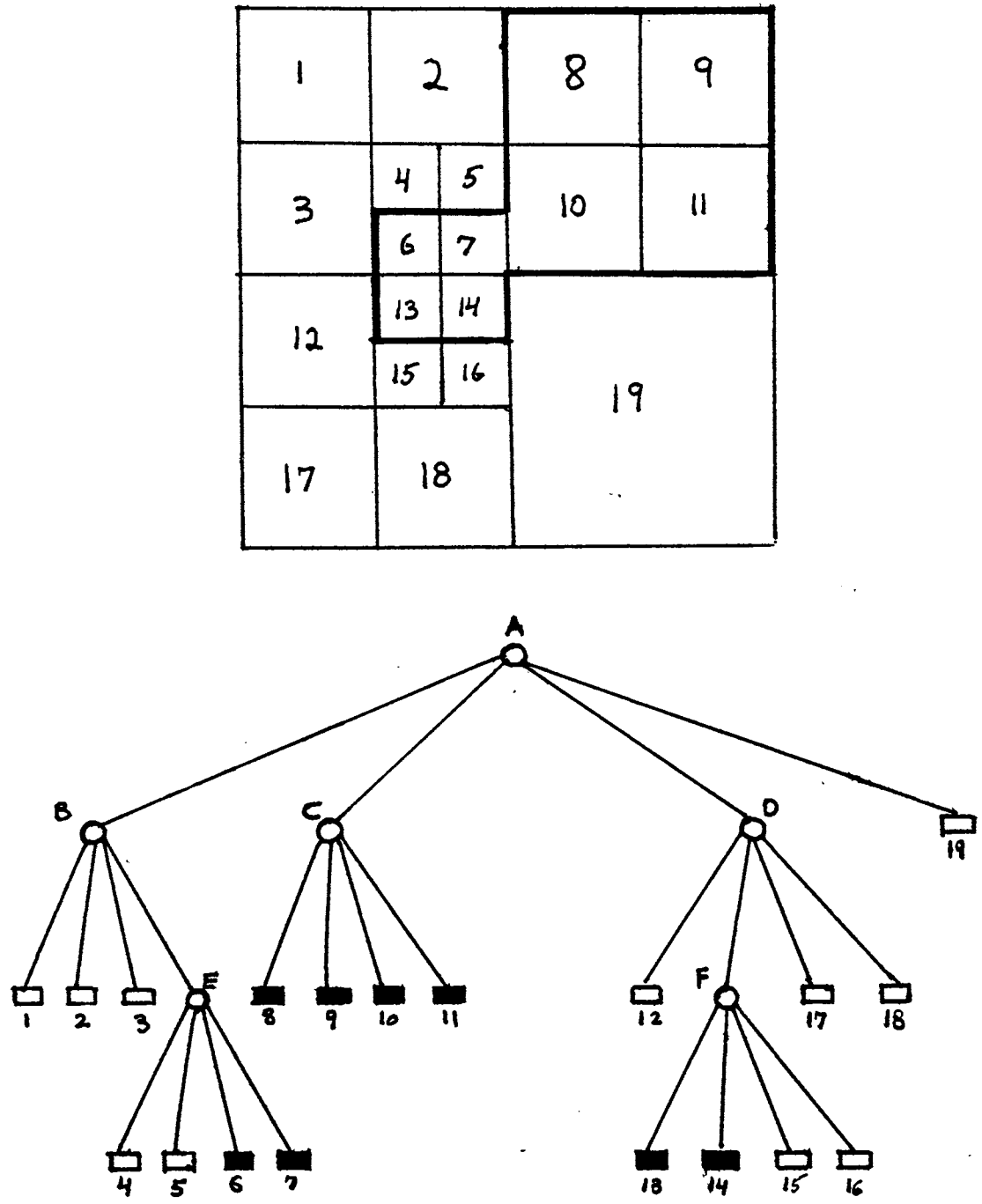
The concept of a forest quadtrees (introduced by Roman [18]) is a decomposition of a quadtree into a collection of subquadtrees. Each subquadtree corresponds to a maximal square. The maximal squares are identified by refining the concept of a nonterminal node that indicate some information

about its subtrees. Samet defines a forest quadtree as follows: "An internal node is said to be of type GB if at least two of its sons are BLACK or of type GB. Otherwise, the node is said to be of type GW" [32].

For example, in Figure 4, nodes C, E, and F are of type GB and nodes, A, B, and D are of type GW. Thus the forest corresponding to Figure 4. is {C,E,F}.

Samet further defined the quadtree concept as follows:

> Each BLACK node or an internal node with a label of GB is said to be a maximal square. A forest is the set of maximal squares that are not contained in other maximal squares and hat span the BLACK area of the image. The elements of the forest are identified by base 4 locational codes. Such a representation can lead to a saving of space since large WHITE items are ignored by it [32].



(Samet, H., "The quadtree and related hierarchical data structures." ACM Comput. Surveys, vol. 16, no. 2, Jun. 1984, pp. 190-235.)

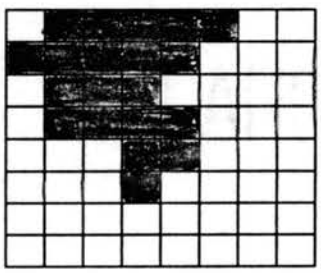
Figure 4. A sample image and its quadtree illustrating the concept of a Forest

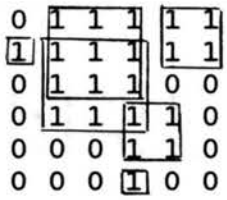
TID Structures

Scott and Iyengar [35] propose a translation invariant data structure (TID). This structure is not effected by translating or rotating an image. The idea is to examine an image and determine the maximal black squares. That is, finding the largest possible squares completely covering an image.

For example, if an image consisted of a 5 x 10 pixel rectangle, two 5 x 5 BLACK squares would completely cover the area; if the rectangle were 5 x 11, however, it would require 3 squares to cover the area (overlapping squares are permitted).

The TID is a list of triples (i, j, s) where (i, j) is the coordinate of the upper left-hand corner of the square and s is the length of the side of the square. The image in Figure 5 is completely defined with six triples:





(0,1,3)
(0,4,2)
(1,0,1)
(1,1,3)
(3,3,2)
(5,3,1)

(Glass, D. J., "Data structures for storing images: region quadtrees." Dep. Comput. Sci., Oklahoma State University, Stillwater, Tach. Rep., osu-cis-tr-87-10, Dec 1987.)

Figure 5. Example of a TID

To shift the image in Figure 5 by (x, y) pixels only requires the following assignment:

for k := 1 to number_of_triples

$(i, j, s)_k := (i + x, j + y, s)_k$

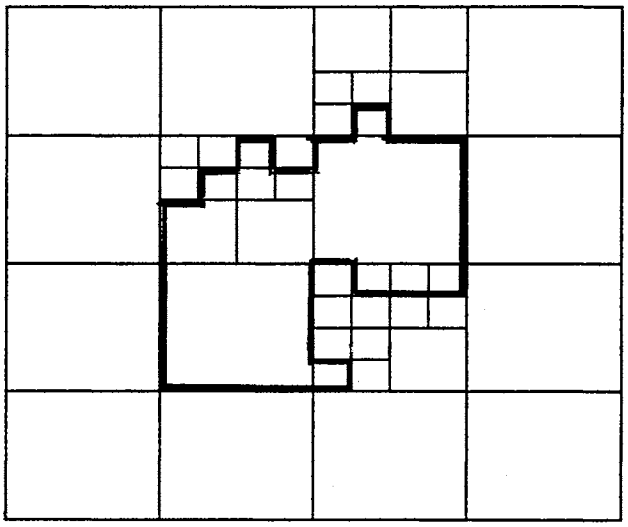
Chain Code

The chain code representation (also known as a boundary or border code) is very commonly used in cartographic applications. As Samet points out in one of his papers, "Chain codes provide a very compact region representation, and make it easy to detect features of the region boundary, such as sharp turns ("corners") or concavities" [5]. On the other hand, it is difficult to perform operations such as union and intersection on a region represented by chain codes.

The chain code can be specified, relative to a given starting point, as a sequence of units vectors (i.e., one pixel wide) in the principal directions. The directions can be represented by numbers.

The following illustrative example is from [5]. Let i, an integer ranging from 0 to 3, represent a unit vector having a direction of 90 x i degrees . Thus, the direction sequence for the boundary of the region in Figure 6, moving clockwise starting from the left of the uppermost border point [5], is

$0\ 3\ 0^2\ 3^5\ 2^3\ 1\ 2\ 3^3\ 0\ 3\ 2^5\ 1^6\ 0\ 1\ 0\ 1\ 0\ 3\ 0\ 1\ 0\ 1$



(Samet, H., "Region representation:boundary codes from quadtrees." Commun. ACM, vol. 23, no. 3, Mar. 1980, pp. 171-179.)

Figure 6. Block Decomposition of a Region

Generalized chain codes, involving more than four directions, can also be used. A general introduction to chain codes can be found in [5].

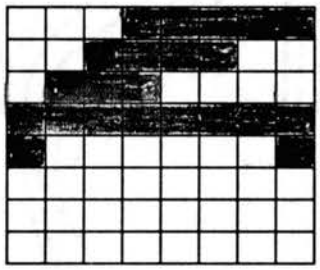
Run Length code

The run length code is an example which makes use of one-dimensional coherence (as opposed to the quadtree encoded image which uses two-dimensional coherence).

A run length encoded image is described in scanline order as a sequence of pairs of values (x,y) where x represents the number of consecutive pixels of value y in each run. According to Wieseman and Oliver "run length encoding gives reasonably good compression in many cases (a

factor of ten may be typical) but the image is not easily manipulated in its coded form" [40]. Therefore, it is particularly useful for image storage and transmission but not much else.

For example, consider the image in Figure 7. The run length encoding is as follows:



3W,5B
2W,4B,2W
1W,3B,4W
8B
1B,6W,1B
24W

Figure 7. An Image and its Run Length Encoding

Treecode

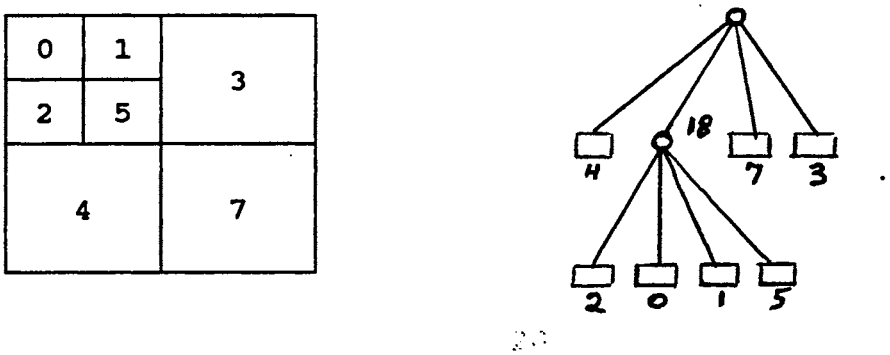
Oliver and Wiseman [40] described a linear code which specifies a quadtree in depth first order. This linear code has found use in several different projects.

Each node, whether non-terminal or leaf, has a value given in 4 bits. One additional bit indicates which sort of node it is. These 5 bit are stored in byte field in memory. A sequence of bytes is read as follows:

1. think of a square area;
2. get next byte;
3. if leaf, color the square with value in 4 bit field;
4. if non-terminal, subdivide the square and return to

step 2 four times for the bottom left, top left, bottom right, and top right squares.

Thus the code 20, 4, 18, 2, 0, 5, 1, 7,3 is understood to represent the quadtree of Figure 8. The non-terminals are 20 and 18 in this case, representing the average value of the whole image (16 + 4) and the top left quadrant (16 + 2), respectively.



(Wiseman, N. E., and Oliver, M. A., "Operations on quadtree encoded images." The Comput. J., vol. 26, no. 1, May 1983, pp. 83-93.)

Figure 8. Quadtree Image from Treecode 20 4 18 2 0 5 1 7 3

Leafcode

Wiseman and Oliver described the leafcode method as follows: "the basic idea in leafcodes is to treat the square image areas represented by the leaves in quadrant as entities separate from the structure of the treecode while retaining the relation to quadtrees by requiring that only square areas corresponding to possible quadtree leaves be

permitted; in general, arbitrary square areas must be broken into the leaves that cover them " [40].

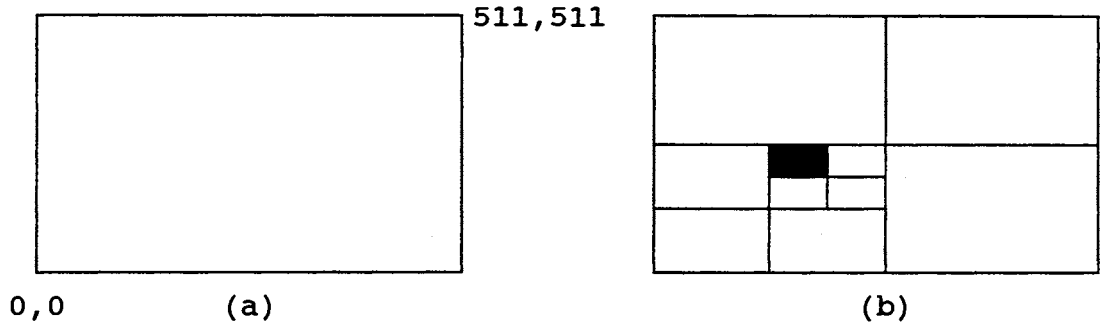
The size and the position of a particular leaf is determined by the recursive structure of the treecode. In leafcodes this information is carried directly in the code for each leaf together with its color. The size of the leaf can be given by the number of pixels along the edge of the square (which will be a power of two) or implicitly by the recursive depth in the quadtree. the position of the leaf is determined by the (x, y) co-ordinates of the pixel in the lower left hand corner of the leaf which will be called the origin of the leaf. Thus, a 512 x 512 picture has a number space as shown in Figure 9a.

The following algorithm determines the specification of the position of a pixel [40]:

1. divide the image into quadrants;
2. choose the quadrant containing the pixel and repeat steps 1, 2 until no further subdivision is possible;
3. write the sequence of quadrant numbers out in base four using 0, 1, 2, 3 to identify the bottom left, top left, bottom right and top right quadrants, respectively.

When the leaves of a treecode are represented in this way, the leaf co-ordinates are seen to be in strictly increasing order. In the example shown in Figure 9b the shaded square has leaf co-ordinates 031 (001101 in binary).

Oliver and Wiseman have found that the space requirement for leafcode can compare favorably with that for the corresponding treecode. For line images on a uniform background the space requirement is then only about 15% greater than for the corresponding treecode.



(Wiseman, N. E., and Oliver, M. A., "Operations on quadtree encoded images." The comput. J., vol. 26, no. 1, May 1983, pp. 83-93.)

Figure ~~9~~ 12. (a) Number space for 512 x 512 picture
(b) Leaf with Leafcode 001101 is shaded

Conversions

The quadtree is proposed as a representation for binary images because its hierarchical nature facilitates the performance of a large number of operations. however, most images are traditionally represented by use of methods such as binary arrays, rasters (i.e., run lengths), and chain codes (i.e., boundaries), some of which are chosen for hardware reasons (e.g., run lengths are particularly useful for rasterlike devices such as television). Techniques are

therefore needed that can efficiently switch between these various representations.

Some of these techniques are described in the following papers:

- treecode into leafcode [40];
- quadtrees from binary arrays [9];
- boundary codes from quadtrees [10];
- quadtrees from boundary codes [11];
- quadtrees to rasters [28].

CHAPTER III

OPERATIONS PERFORMED ON QUADTREES

Set Operations

The quadtree is especially useful for performing set operations such as the union (i.e., overlay) and intersection of several images. The ability to perform set operations quickly is one of the primary reasons for the popularity of quadtrees over alternative methods.

Intersection

The intersection operation involves traversing two given quadtrees as follows: when one tree has a son that is a BLACK leaf, while the other has a corresponding son that is not BLACK, the BLACK leaf is replaced by the corresponding subtree. If one tree has a leaf that is WHITE, the intersection tree will have a corresponding WHITE leaf. Finally, if both trees have GRAY nodes in corresponding positions, the nodes' sons are examined recursively, using the same process. Figure 10d illustrates the intersection of Figures 10a and 10b. (See Appendix A for detailed procedure).

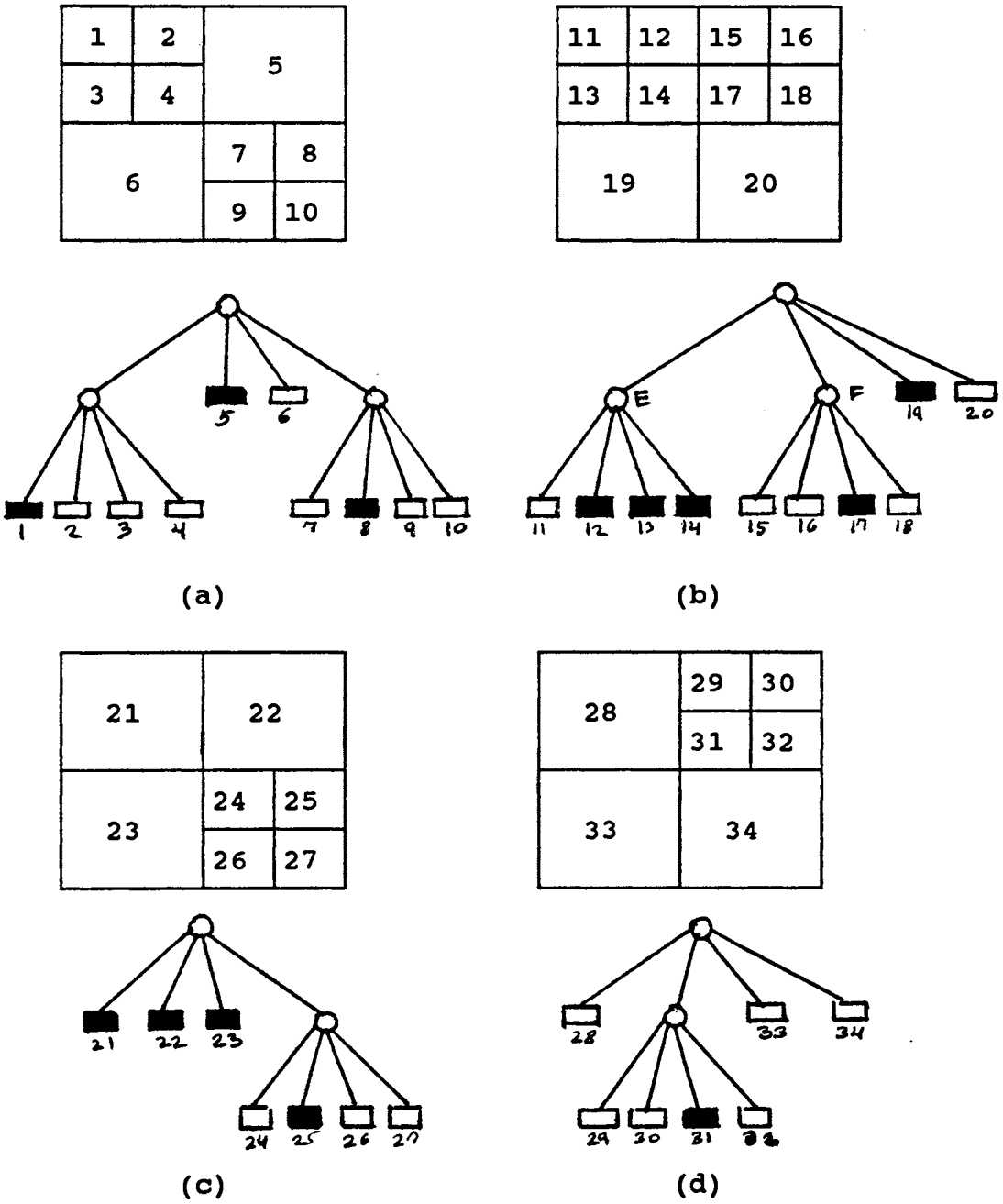
Union

The union operation is very similar to the intersection operation. The two given quadtrees are traversed in parallel as follows: when a BLACK leaf is encountered, this becomes the subtree at the current position in the union tree. A WHITE leaf in one tree results in the corresponding subtree of the other tree becoming the subtree at the current position of the union tree. When there are two GRAY nodes, the procedure is called recursively for the subtrees rooted at these nodes.

Figure 10c illustrates the union of figures 10a and 10b (see Appendix A for detailed procedure).

Complement

The complement operation is a very simple operation in a quadtree, and does not change the structure of the tree at all. It involves changing BLACK nodes (or pixels) into WHITE, and WHITE nodes (or pixels) to BLACK. (see Appendix A for detailed procedure).



(Samet, H., "The quadtree and related hierarchical data structures." ACM Comput. Surveys, vol. 16, no. 2, Jun. 1984, pp. 190-235.)

Figure 10a. Image and its Quadtree

Figure 10b. Image and its Quadtree

Figure 10c. Union of the images in Figures 10a and 10b

Figure 10d. Intersection of the images in Figures 10a and 10b

Geometric Transformations

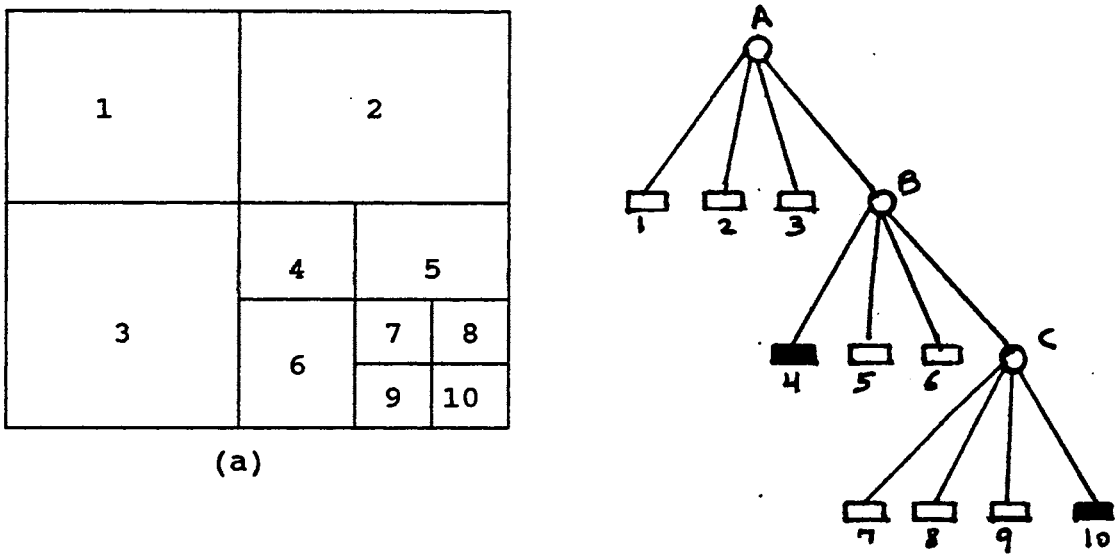
As Samet points out in one of his papers, "One of the primary motivations for the development of the quadtree concept is a desire to provide an efficient data structure for computer graphics" [32]. Therefore, one needs to develop a system that has the capability of performing a number of basic transformations.

Rotation

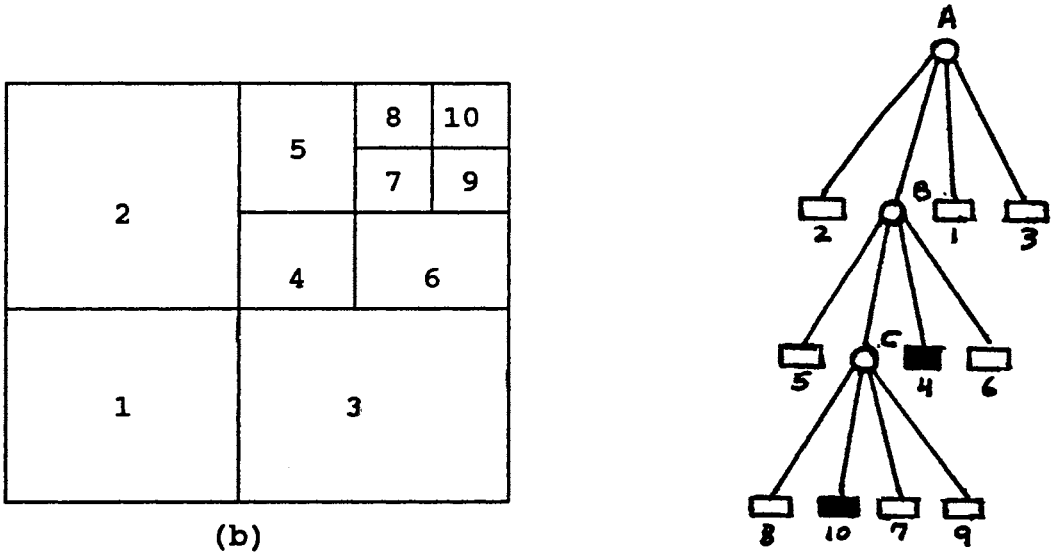
Rotation by multiples of 90 degrees is quite simple. The operation involves a recursive rotation of sons at each level of the quadtree. Figure 11a and Figure 11b illustrate the rotation operation. Figure 11b is the result of rotating Figure 11a by 90 degrees counterclockwise. Notice how the NW, NE, SW, SE sons have become SW, NW, SE, and NE sons, respectively, at each level in the quadtree (see Appendix A for detailed procedure).

Scaling

Scaling by a power of two is also a simple operation when using quadtrees. When traversing the quadtree from the root down to its leaves, the resolution is increased by power of two at each level down. Therefore, we can control the size of the image by simply determining the size at the root level (remember that the size is a power of two).



(a)



(b)

(Samet, H., "The quadtree and related hierarchical data structures." ACM Comput. Surveys, vol. 16, no. 2, Jun. 1984, pp. 190-235.)

Figure 11. Rotating (a) by 90 degrees counterclockwise yields (b)

Windowing

Windowing is another operation that is useful in graphics applications. The process involves extracting a rectangular window from an image represented by a quadtree and building a quadtree for the window. Rosenfeld et al. [19 as cited in 22] introduced an algorithm that extracts a square window of size 2^k by 2^k at an arbitrary position in a 2^n by 2^n image. Samet gives the following explanation:

> In essence, the new quadtree is constructed as the input quadtree is decomposed and relevant blocks are copied into the quadtree. The execution time of this process depends both on the relative position of the center of the window with respect to the center of the input quadtree, and the sizes of the blocks in the input quadtree that overlap the window. For rectangular windows, windowing is simple to implement if the squarcode representation of Wiseman and Oliver is used [32].

Computation

Area

In order to find the area of an image represented by a quadtree, it is necessary to traverse the quadtree in postorder and accumulate the sizes of the BLACK blocks.

Samet calculates the area as follows: "Assume that the root of a 2^n by 2^n image is at level n and the number of pixels in such an image is 2^{2n} for a BLACK block at level k, the contribution to the area is 2^{2k} " [32]. (See Appendix A for detailed procedure).

Perimeter

Computing the perimeter of an image represented by a quadtree can be carried out as follows: A postorder tree traversal is performed, and for each BLACK node that is encountered its four adjacent sides are explored in the search for adjacent WHITE nodes. Then, for each adjacent WHITE node that is found the length of the corresponding shared side is included in the perimeter.

The algorithm involves a certain amount of duplication because each adjacency between two BLACK blocks is explored twice, and neither of these adjacency explorations contributes to the value of the perimeter. Samet [32] suggested an alternative algorithm that performs adjacency exploration only for southern and eastern neighbors. That is, for each BLOCK node a search is made for adjacent WHITE southern and eastern neighbors, and for each BLACK southern and eastern neighbors. But the problem with such a method is that the northern and western boundaries of the image are never explored. The problem can be solve by embedding the image in a white region, as shown in Figure 12.

(Samet, H., "The quadtree and related hierarchical data structures." ACM Comput. Surveys, vol. 16, no. 2, Jun. 1984, pp. 190-235.)

Figure 12. An Image totally surrounded by background

Samet claimed that both formulations of the algorithm have expected execution times that are proportional to the total number of nodes in the quadtree.

Centroid

The centroid of a binary image is a point (x,y) such that $\bar{x}$ is the average value of the x-coordinates of all the BLACK points of the image and $\bar{y}$ is the average of the y-coordinates of the BLACK points. In other words, if there are m BLACK points in the image, $(x_1,y_1),\ldots,(x_m,y_m)$, the centroid is

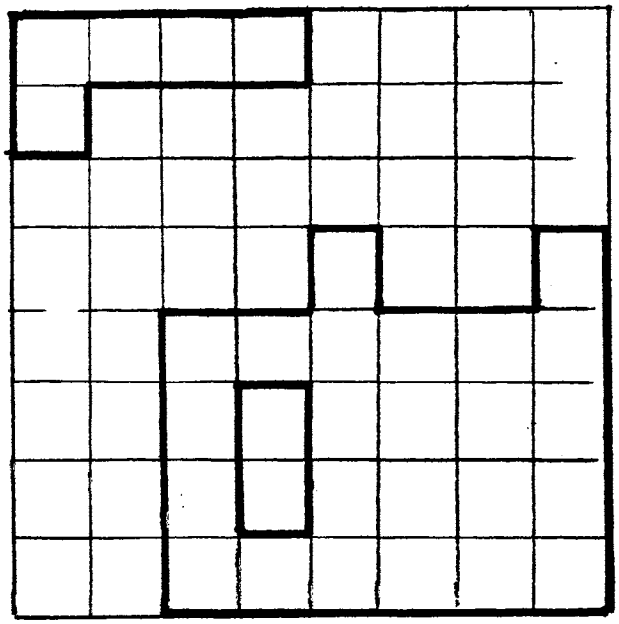
$$(\bar{x},\bar{y})=(\Sigma x_i/m,\Sigma y_i/m).$$

The centroid procedure can be found in [36].

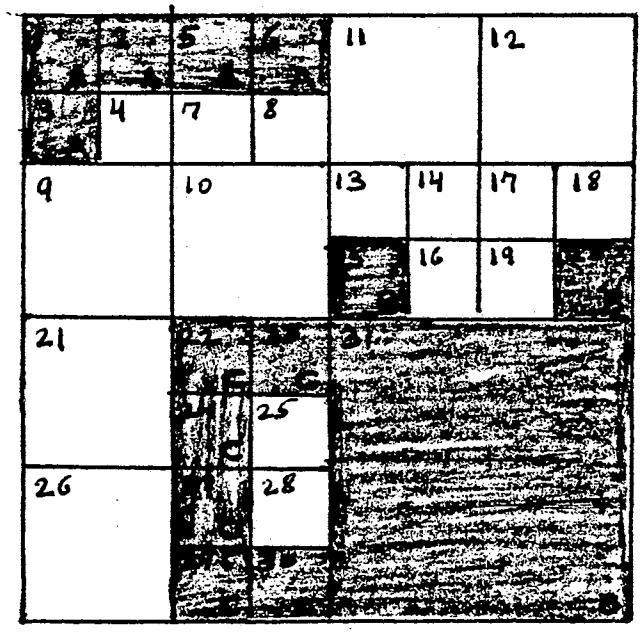
Connected Component Labeling

"Connected component labeling is one of the basic operations of an image-processing system" [32]. The process is analogous to finding the connected components of a graph. Consider the image of Figure 13 which has two components. A common method for performing this process [20], is the "breadth-first" approach: given a binary representation of an image, the image is scanned row by row from left to right and the same label is assigned to adjacent BLACK pixels that are found to the right and in the downward direction. During this process, pairs of equivalences may be generated, thus two more steps are needed: one to merge the equivalences and the second to update the labels associated with the various pixels to reflect the merger of the equivalences.

Samet [21] uses a quadtree to perform the same operation. The algorithm involves an analogous three-step process. Samet claimed [32] that the algorithm has an average execution of O(B log B), where B is the number of black nodes in the quadtree that represent the image.



(a)



(b)

(Samet, H., "The quadtree and related hierarchical data structures." ACM Comput. Surveys, vol. 16, no. 2, Jun. 1984, pp. 190-235.)

Figure 13. (a) An Image
(b) Block Decomposition of the image in (a)

Top-Down Quadtree Traversal

Many standard image processing operations can be implemented using quadtrees as a simple tree traversal. Computation is performed at each terminal node involving some of that node's neighbors. Most of these operations involve the use of bottom-up neighbor-finding techniques which search for a nearest common ancestor.

Several top-down techniques have been proposed which make use of a neighbor vector as the tree is traversed. A simplified version of the top-down method for a quadtree can been found in a paper by Samet [33]. Samet claims that his algorithm differs in part from prior work in its ability to compute diagonally adjacent neighbors rather than just horizontally and vertically adjacent neighbors. The algorithm builds a neighbor vector for each node using a minimal amount of information. Analysis of the algorithm shows that its execution time is directly proportional to the number of nodes in the tree. However, it does require some extra storage. As stated by Samet, "Use of the algorithm leads to lower execution time bounds for some common quadtree image processing operations such as connected component labeling" [33].

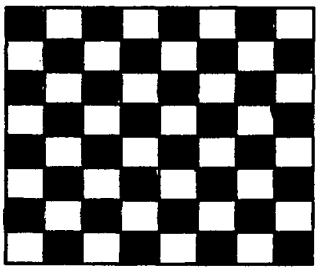
The Space Efficiency of Quadtrees

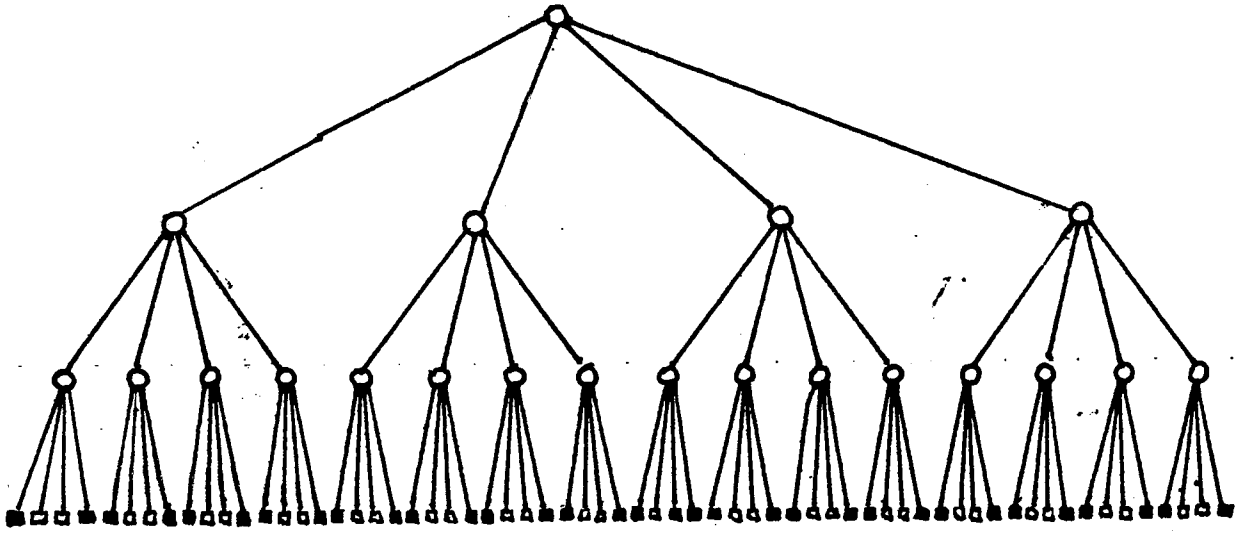
The problem of space efficiency has been a crucial factor in the development of quadtrees. According to Samet:

> The prime motivation for the development of the quadtree has been the desire to reduce the amount of space necessary to store data through the use of aggregation of homogeneous blocks [32].

But the quadtree is not always the ideal representation. The worst case for a quadtree of a given depth in terms of storage requirements occurs when the region corresponds to a checkerboard pattern as shown in Figure 14. The number of nodes in the quadtree is obviously a function of the number of levels in the quadtree (i.e., the resolution).

A tree implementation of a quadtree has overhead in terms of the number of internal nodes. Samet [32] claimed that for an image with B and W BLACK and WHITE blocks, respectively, (4 / 3) (B + W) nodes are required. A binary array representation of a 2^n by 2^n image requires only 2^{2n} bits; however, this quantity grows quite quickly. Furthermore, if the amount of aggregation is minimal (e.g., a checkerboard image; one leaf node for every pixel), then the quadtree is not very efficient. Pointerless representations, such as linear quadtree and the DF-expression, are used to avoid the overhead of a large number of internal nodes. In fact, the DF-expression requires at most two bits per node.





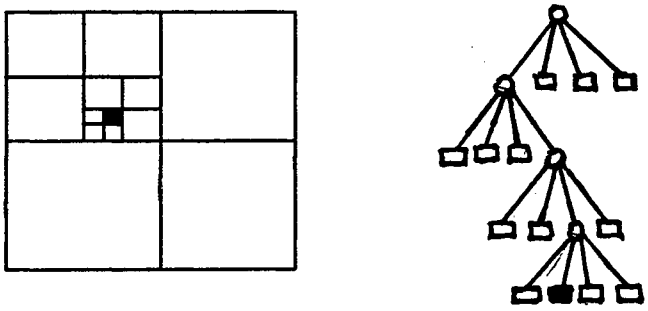
(Samet, H., "The quadtree and related hierarchical data structures." ACM Comput. Surveys, vol. 16, no. 2, Jun. 1984, pp. 190-235.)

Figure 14. A checkerboard and its quadtree

The main disadvantage of quadtrees is that they are shift variant. That is, two identical regions differing only by a translation in an image may have quite different quadtrees. The following example is illustrated in a paper by Dyer [3]. A 2^m by 2^m square region may be represented by a single node or as many as $O(2^m)$ nodes depending on its position in the image. Dyer investigated the best, average, and the worst case quadtree encoding efficiencies of a 2^m by 2^m region in a 2^n by 2^n image.

Dyer claims that the best case occurs when the region can be represented by a single BLACK node at level m, and that only $O(n - m)$ nodes are required when the region is in any of 2^{n-m+1} positions.

The worst case occurs when shifting the region to the right and down one pixel from the best case. Dyer [3] has shown that the average case required $O(2^{m+2} + n - m)$ quadtree nodes.



(Dyer, C. R., "The space efficiency of quadtrees." Comput. Gr. Image Process., vol. 19, no. 4, Aug. 1982, pp. 335-348.)

Figure 15. Best case position of a 2^m by 2^m region in a 2^n by 2^n binary image

Pyramids

Samet defines a pyramid as follows: "Given a 2^n by 2^n image array, say A(n), a pyramid is a sequence of arrays {A(i)} such that A(i-i) is a version of A(i) at half the resolution of A(i) is a single pixel" [32]. Pyramid can also be defined in a more general way by permitting finer scales of resolution than the power of two scale.

Giving a 2^n by 2^n image, a recursive decomposition into quadrants is performed, just as in quadtree construction. The only difference is that we keep subdividing until we reach the individual pixels. The leaf nodes of the resulting tree represent the pixels, whereas the nodes immediately above the leaf nodes correspond to the array A(n-1), which is of size 2^{n-1} by 2^{n-1}. The nonterminal nodes are assigned a value that is a function of the nodes below them such as the average GRAY level.

The above definition of a pyramid is based on nonoverlapping 2 by 2 blocks of pixels. The difference between pyramids and quadtrees, is stated by Samet as follows:

> Pyramids and quadtrees, although related, are different entities. A pyramid is a multiresolution representation, whereas the quadtree is a variable resolution representation. Another analogy is that pyramid is a complete quadtree [32].

CHAPTER IV

PROBLEM DESIGN

Objectives

Chapters II and III described the concepts of using quadtrees to represent an image and various operations performed on such a data structure. This chapter describes the program design and implementation and how some of the concepts discussed in Chapters II and III were implemented. Analysis and comparisons are also included.

Since the quadtree can be used to represent an image, the idea is to build and store each consonant and vowel from the Hebrew language in a quadtree. Then, some of the operations described in chapter III (e.g., scaling) can be implemented on these quadtrees.

Program Design and Implementation

Terms

Before presenting the program design and implementation, one needs to understand the following terms:

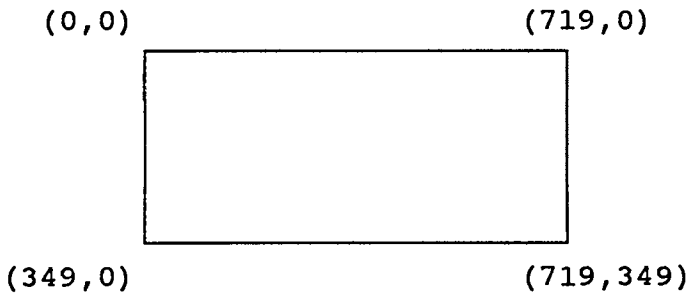
- pixel
- screen

. object-oriented and bit-mapped images

pixel. The term pixel is an acronym for picture element. Pixels, in fact, are the basic elements that make up a video display. The pixels are combined to make the text and graphic images on the computer monitor. Pixels can be displayed as black or white with a monochrome card, or in any color supported by a color card.

screen. A screen is the configuration of pixels that make up displayed text or graphic images. Depending on the type of graphics card installed in the microcomputer system, the screen display will be made up of different horizontal-by-vertical pixel dimensions. The Amdek system was configured with the Hercules (Hercules Technology, 2550 Ninth St., Berkeley, CA 94710) graphics card.

Note that by convention, the upper left corner of the graphics screen is (0,0). Thus, in Hercules mode, the screen look like this:



Object-Oriented and Bit-Mapped Images. Graphics programs create two types of images: object-oriented (sometimes called vector graphics) and bit-mapped images. An object-oriented graphic is built from lines and shapes; a bit-mapped image is composed of dots (i.e., pixels).

Implementation Steps

As mentioned before, the main goal is to build a quadtree for each consonant and vowel. To accomplish this, the following steps were taken:

1. drawing the image (i.e., consonant or vowel) on the screen using any available graphics software
2. building a complete quadtree
3. scanning the image pixels in a predetermined order and "coloring" the corresponding quadtree's leaf nodes
4. merging groups of four pixels or four blocks of a uniform color
5. saving the resulting quadtree constructed in step four above

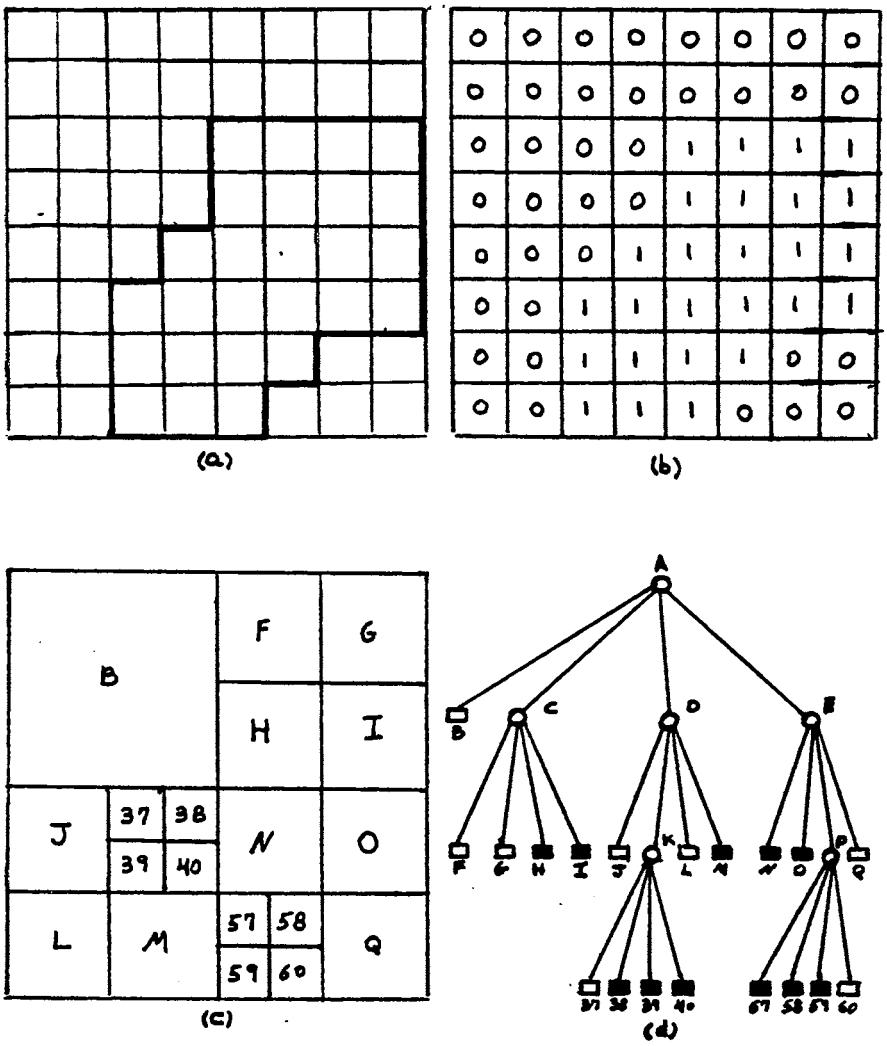
A detailed description is given below for each step stated above.

Step 1. Drawing the image. Each consonant or vowel is drawn using some available graphics software. The consonants and vowels are subsequently imported into the program which scans and builds the corresponding quadtrees. Note that the language being used (i.e., Turbo Pascal) must be able to import the images drawn by the graphic software (as previously mentioned, graphic programs create two types of images: object-oriented and bit-mapped images).

The image must be displayed in the upper left corner of the graphics screen at coordinates (1,1). This restriction is due to the fact the image is scanned starting at coordinate (1,1). This coordinate was selected only for ease of implementation.

step 2. Building a complete quadtree. The binary array is probably the most common method to represent an image. There are many methods to construct a quadtree from a binary array [22]. The simplest approach is one which converts the array to a complete quadtree.

For example, consider the image in Figure 1a, repeated for ease of reference on the following page. The image size is 2^3 by 2^3 pixels (i.e., 64 pixels). The binary representation and the corresponding complete quadtree of the image are illustrated in Figure 16.

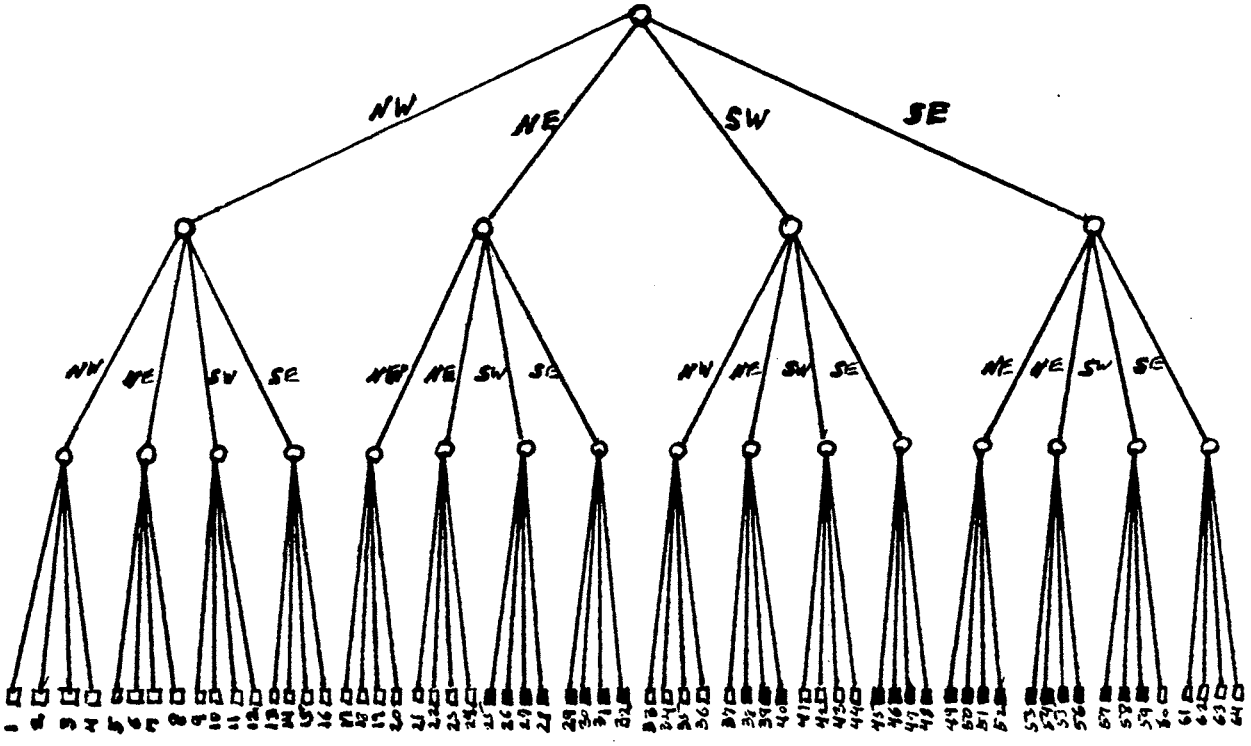


(Samet, H., "The quadtree and related hierarchical data structures." ACM Comput. Surveys, vol. 16, no. 2, Jun. 1984, pp. 190-235.)

Figure 1. (a) A region (b) Binary array (c) Block decomposition of the region (d) Quadtree representation of the blocks in (c).

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

(a)



(b)

Figure 16. (a) Binary representation of the image in Figure 1a
(b) Complete quadtree of the image in (a)

Thus, a complete quadtree has a leaf node for every pixel. In general, for a 2^n by 2^n image, a complete quadtree is of height n with 2^{2n} leaf nodes.

Step 3. Scanning the image pixels. The image is scanned in the order defined by the labels in Figure 16a. This order corresponds to a postorder traversal of the quadtree in Figure 16b. That is, the nodes of the quadtree are visited recursively in the following order: NW, NE, SW, and SE.

The image is displayed on the screen starting at coordinate (1,1) (as stated earlier, this coordinate was chosen for ease of implementation). Thus, label 1 of Figure 16a corresponds to coordinate (1,1), label 2 corresponds to coordinate (1,2), label 3 corresponds to coordinate (2,1), and so on.

In order to generate the above coordinates, a special function was developed (see Appendix A). The function stores the coordinates in an index file that was used by the program performing the scanning. One should note, that, in fact, there is no need to generate all the coordinates. It is enough to generate only the coordinates correspond to the NW leaf nodes.

For example, if coordinate (1,1) is given (i.e., label 1 in Figure 16a), it is easy to compute labels 2, 3, and 4 as follows:

let coordinate (1,1) corresponds to (x,y), then

coordinate (1,2) (i.e., label 2) = (x,y+1) = (1,2)

coordinate (2,1) (i.e., label 3) = (x+1,y) = (2,1)

coordinate (2,2) (i.e., label 4) = (x+1,y+1) =(2,2)

Therefore, for the region of Figure 16a, only the following coordinates need to be generated:

1,1		1,3		1,5		1,7	
3,1		3,3		3,5		3,7	
5,1		5,3		5,5		5,7	
7,1		7,3		7,5		7,7	

Figure 17. Scanning coordinates

Note that the scanning region must be of size 2^n by 2^n (i.e., powers of two). Therefore, the image must be displayed within a predetermined region.

The "coloring" (i.e., scanning) process is as follows: starting with the root node, the complete quadtree is recursively traversed in postorder. Whenever a leaf node is encountered, its corresponding pixel color is obtained by

using the corresponding coordinate in the index file. The process is terminated when all leaf nodes are visited (see Appendix A for the scanning function).

Step 4. Merging groups of four pixels of Uniform Color. At this point, the "coloring" process of the complete quadtree is completed. Each leaf node of a complete quadtree corresponds to exactly one pixel of the image. As mentioned in Chapter II, the quadtree is a collection of maximal blocks that partition a given region (note that the blocks may possibly overlap). The emphasis is on maximal blocks. Therefore, the purpose of step four, is to merge groups of four pixels or four blocks of a uniform color, until no further merging is possible. Therefore, the merging process may reduce the number of nodes in the quadtree significantly.

For example, the following leaf nodes of the complete quadtree in Figure 16b would be merged:

nodes 1, 2, 3, and 4 (all four nodes are WHITE)

nodes 5, 6, 7, and 8 (all four nodes are WHITE)

nodes 9, 10, 11, and 12 (all four nodes are WHITE)

Upon completion of step four, the resulting quadtree is given in Figure 1d on page 45. Observe, that the number of nodes in the quadtree of Figure 1d is 25. This is a savings of over 40 percent over the total number of nodes (i.e., 64 nodes) in the original complete quadtree.

Step 5. Saving the resulting quadtree. To save the quadtree constructed in step four, the DF-expression method described in Chapter II is implemented. When saved in a file, the quadtree can then be reconstructed from the DF-expression (see Appendix A for detailed functions).

Software Development

The purpose for developing the software, was to familiarize a beginning student with the Hebrew consonants and vowels. The software contains about 2000 lines of code in Turbo Pascal, and 7 new functions related to quadtrees (see Appendix A). Four months of designing, coding, and debugging were spent.

The software makes use of the quadtree theory by representing each consonant and vowel in a specific quadtree. For each consonant and vowel a corresponding quadtree was built and stored (as explained in the implementation section of this chapter). When executed, the program automatically reconstructs the original quadtree for each consonant and vowel from its corresponding file. Note that the storage for these quadtrees is dynamically allocated during program execution. At termination, these quadtrees no longer exist and the storage space used by these quadtrees is returned to the system's free storage space. Appendix B contains tables indicating the storage requirements for each file and for each quadtree. Also included are various screens generated by the software.

Analysis and Comparison

Region representation is an important issue in image processing. As stated in Chapter III, the prime motivation for the development of the quadtree has been the desire to reduce the amount of space required to store images. It was also explained that the quadtree is not always the ideal representation.

The approach in this thesis was to convert the image binary array to a complete quadtree (i.e., one node per pixel) then to reduce the quadtree size through repeated attempts at merging groups of four pixels or four blocks of a uniform color. The major disadvantage of this approach is the extreme waste of storage space, because many nodes may be created needlessly. In fact, the complete quadtree may not fit in the available memory, whereas the resulting quadtree may fit. In particular, for a 2^n by 2^n image, 2^{2n} BLACK and WHITE nodes (i.e., leaf nodes), and an additional (BLACK + WHITE - 1)/3 GRAY nodes [32] (i.e., nonterminal nodes) are needed to construct the corresponding complete quadtree. This is clearly undesirable when compared with a maximum of 2^{2n} bits required by the binary array representation.

The minimum storage requirements for a complete quadtree is analyzed as follows. Each node has at least five fields. Four pointer fields, one to each son-quadrant (i.e., NW, NE, SW, and SE), and one field for the node color

(i.e., BLACK, WHITE, or GRAY). That is, at least five bytes per node.

Lets us assume that an image of size 2^8 by 2^8 is given. To construct a complete quadtree for this image, the following storage is needed:

leaf nodes: 256 x 256 x 5 = 327,680 bytes

internal nodes: 1/3 x 327,680 = 109,227 bytes (see last page)

Thus, about 437 KB is required. Some microcomputer systems are not equipped to handle such amount of memory requirement.

In addition to the extreme waste of storage, the merging process involves an extra overhead. Thus, the complete quadtree approach may be simple, but obviously it has some major disadvantages. Therefore, a better approach should be used.

Consider the following approach. The elements of the binary array are visited in the order defined by Figure 16a. However, in order to avoid the needless creation of nodes in the case of the complete quadtree, a leaf node is created only if it is known to be maximal. Samet points out an analogous situation: "An equivalent statement is that the situation does not arise in which four leaves of the same color necessitate the changing of the color of their parent from GRAY to BLACK or WHITE as is appropriate" [32].

For example, consider pixels 25, 26, 27, and 28 in Figure 16a. All these pixels are BLACK, therefore, only one node should be created for these four pixels. That is, node H in Figure 1d on page 45.

A similar approach is presented by Hanan Samet [28]. He developed an algorithm for converting rasters to quadtrees. That is, obtaining an in-core quadtree representation given the row-by-row description of a binary array. Thus, the pixels of the image of Figure 1a on page 45 would be visited in the order defined by the labels on the array of Figure 18. Samet stated in one of his papers that "One of the algorithm's key features is that at any instant of time (i.e., after each pixel in a given row has been processed) a valid quadtree exists with all unprocessed pixels presumed to be WHITE" [28]. That is, as the quadtree is built, nodes are merged to yield maximal blocks.

Samet has shown that the algorithm's execution time has time complexity proportional to the number of pixels in the image. The algorithm is also space efficient because merging is attempted whenever possible. That is, after processing each pixel in a row, the resulting quadtree contains a minimal number of nodes.

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

(Samet, H., "The quadtree and related hierarchical data structures." ACM Comput. Surveys, vol. 16, no. 2, Jun. 1984, pp. 190-235.)

Figure 18. Raster labeling

Consonant Representation

The software developed, used the quadtree representation for displaying the Hebrew consonants and vowels. The question which might arise is whether there are alternative methods to represent the alphabet other than the quadtree representation. For purposes of comparison, consider the bitmap method discussed below.

Bitmap Representation

The bitmap is a complete digital representation of an image. Each pixel in the image corresponds to one or more bits in the bitmap. Monochrome bitmaps require only one bit per pixel whereas color bitmaps require additional bits to indicate the color of each pixel.

Bitmaps have two major drawbacks. First, they are highly sensitive to problems involving device independence, of which the most obvious is color. Displaying color bitmap on a monochrome device is often unsatisfactory. Another problem is that although bitmaps can be stretched or compressed, this generally involves duplicating or dropping rows or columns of pixels and can lead to distortion in the scaled image. The second major drawback of bitmaps is that they require a large amount of storage space. For instance, a bitmap representation of an entire 640-by-350, eight-color EGA screen requires 84 KB.

For a monochrome bitmap, the format of the bits is relatively simple and can be derived almost directly from the image to be created. For instance, consider the Hebrew consonant BET on the next page:

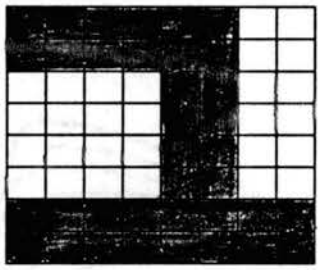


Figure 19. The Hebrew consonant BET

The consonant BET above, can be represented as a series of bits (0 for BLACK and 1 for WHITE). Reading these bits from left to right one can then assign each group of 8 bits a hexadecimal byte. If the width of the bitmap is not a multiple of 8, the bytes are padded to the right with zeros to get an even number of bytes. Thus, the bitmap representation of the consonant BET is:

```
1 1 1 1 1 1 0 0 = FC
1 1 1 1 1 1 0 0 = FC
0 0 0 0 1 1 0 0 = 0C
0 0 0 0 1 1 0 0 = 0C
0 0 0 0 1 1 0 0 = 0C
0 0 0 0 1 1 0 0 = 0C
1 1 1 1 1 1 1 1 = FF
1 1 1 1 1 1 1 1 = FF
```

Thus, 8 bytes are required to store the consonant BET in a file.

Now, consider the quadtree in Figure 20 for the same consonant BET. It has 21 nodes, thus 21 bytes are required to store the quadtree using the DF-expression, and additional 105 bytes (21 x 5, assuming 5 bytes per node) need to be dynamically allocated to construct the quadtree.

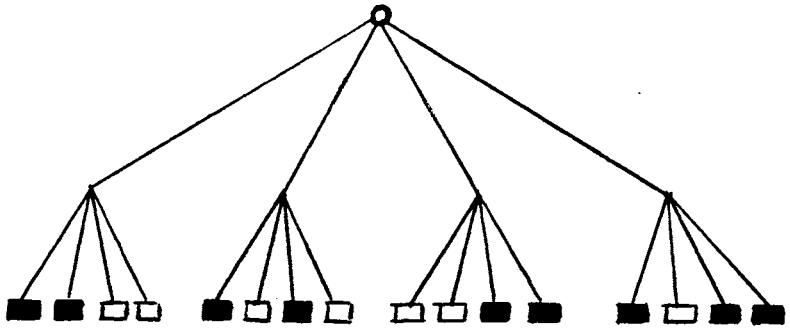


Figure 20. Quadtree for the consonant BET in Figure 19

From the example above it seems that the bitmap representation is better spacewise than the quadtree representation. But this is not always true. Given a different image size and shape, the quadtree representation might be better. For example, if the image in Figure 19 was all black, a quadtree of only one node would represent the image, and only 5 bytes would be needed to store the quadtree.

CHAPTER V

SUMMARY AND CONCLUSIONS

This thesis was logically divided into two parts. In the first part, functions that scan a given image and build its corresponding quadtree were developed. In part two, a software, whose purpose it is to familiarize a beginning student to the Hebrew consonants and vowels, was developed. The software makes use of the functions developed in the first part.

Summary

At the onset of the thesis, the complete quadtree approach was used. This approach is probably the simplest one, but it has two major drawbacks. The extreme waste of storage space (i.e., the needless creation of nodes), and the additional overhead that the merging process creates.

As to operations performed on quadtrees, the fact that an image can be scaled only by powers of two and rotated by multiples of 90 degrees, may prevent some application programs from using the quadtree representation. As opposed to the complete quadtree approach, two other approaches were

given. Both methods avoid the needless creation of extra nodes.

Conclusions

As to representing an image by a quadtree, it was mentioned in Chapter II that there are alternative methods to represent quadtrees. Each method has its strong and weak points regarding time complexity and space requirement. The ease of performing operations on quadtrees, such as rotation, may vary from one method to another.

In Chapter IV, the bitmap representation was compared with the quadtree representation. The comparison example supports the conclusion that an analysis is required before choosing one method over the other. Thus, when searching for a particular method to represent an image, the following points should be taken into consideration:

- the image size;
- storage space required to store the image;
- operation performed on the image;
- monochrome or color representation;
- execution time.

Suggested Future Work

As explained in Chapter IV, the scanning process employed postorder traversal of the complete quadtree. For each leaf node, its corresponding pixel color was obtained.

To do so, a special index file was created, for mapping each leaf node to its corresponding pixel (i.e., coordinate).

It would be an interesting topic to develope a function (i.e., formula) that will generate these coordinates. That is, a function to map each label in Figure 16a to its corresponding coordinate in Figure 17. Thus saving the storage required to store the index file.

An alternative topic is to scan the image binary array by rows or by columns (or any other order) and then to develope an algorithm to traverse the complete quadtree in that order.

In Chapter III, it was explained that the main disadvantage of the quadtree is that it is shift-variant. That is, shifting a given image may significally change the size of the corresponding quadtree. A potential research topic would be to develop an algorithm that will determine the best location of an image in a given region, so that the corresponding quadtree constructed would have minimum size.

SELECTED BIBLIOGRAPHY

[1] Burt, P. J., "Tree and pyramid structures for coding hexagonally sampled binary images." Comput. Gr. Image Process., vol. 14, no. 3, Nov. 1980, pp. 249-270.

[2] Burton, F. W., "Comment on the explicit quadtree as a structure for computer graphics." Comput. J., vol. 26, no. 2, May 1983, p. 188.

[3] Dyer, C. R., "The space efficiency of quadtrees." Comput. Gr. Image Process., vol. 19, no. 4, Aug. 1982, pp. 335-348.

[4] Finkel, R. A., and Bentley, J. L., "Quadtrees: a data structure for retrieval on composite keys." Acta Informatica, vol. 4, Apr. 1974, pp. 1-9.

[5] Freeman, H., "Computer processing of line drawing images." ACM Comput. Surveys., vol. 6, no.1, Mar. 1974, pp. 57-97.

[6] Garantini, I., "Translation, rotation, and superposition of linear quadtree." Int. J. Man-Mach. Stud., vol. 18, no. 3, Mar. 1983, pp. 253-263.

[7] Garantini, I., "An effective way to represent quadtrees." Commun. ACM, vol. 25, no. 12, Nov. 1982, pp. 905-910.

[8] Glass, D. J., "Data structures for storing images: region quadtrees." Dep. Comput. Sci., Oklahoma State University, Stillwater, Tech. Rep., OSU CIS TR-87-10, Dec. 1987.

[9] Grosky, W. I., "Optimal quadtrees for image segments." IEEE Trans. Pattern Analysis Mach. Intelligence, vol. 5, no. 1, Jan. 1983, pp. 68-74.

[10] Hunter, G. M., "Properties and application of pictures represented by quadtrees." Comput. Gr. Image Process., vol. 10, no. 3, Jul. 1979, pp. 289-296.

[11] Jones, L., "Space and time efficient virtual quadtrees."
IEEE Trans. Pattern Analysis Mach. Intelligence, vol. 6, no. 2, Mar. 1984, pp. 244-247.

[12] Kawaguchi, E., "On the method of binary-picture representation and its application to data compression."
IEEE Trans. Pattern Analysis Mach. Intelligence, vol. PAMI-2, no. 1, May. 1980, pp. 27-35.

[13] Kawaguchi, E., "DF-expression of binary-valued picture and its relation to other pyramidal representations."
In Proceeding of the 5th International Conference on Pattern Recognition (Miami Beach, Fla, Dec.). IEEE, New York 1980, pp. 822-827.

[14] Kawaguchi, E., "Depth-first expression viewed from digital picture processing."
IEEE Trans. Pattern Analysis Mach. Intelligence, vol. 5, no.4, Jul. 1983, pp. 374-384.

[15] Klinger, A., "Organization and access of image data by areas."
IEEE Trans. Pattern Analysis Mach. Intelligence, vol. 1, no. 1, Jan. 1979, pp. 50-60.

[16] Milford, D. J., "Quad encoded display."
IEE Proceedings, vol. 131, no. E3, May 1984, pp. 70-75.

[17] Nagy, G., "Geographic data processing."
ACM Comput. Surveys., vol. 11, no. 2, Jun. 1979, pp. 139-181.

[18] Raman, V., "Properties and applications of forest quadtrees for picturial data representation."
BIT, vol. 23, no. 4, 1983, pp. 472-486.

[19] Rosenfeld, A., and Pfaltz, J. L., "Sequential operations in digital image processing."
J. ACM, vol. 13, no. 14, Oct. 1966, pp. 471-494.

[20] Rosenfeld, A., Samet, H., Shaffer, C., and Webber, R. E., "Application of hierarchical data structures to geographical information systems." TR-1197, Comput. Sci. Dept., University of Maryland, College Park, Md., Jun. 1982.

[21] Samet, H., "Connected component labeling using qudtrees." J. ACM., vol. 28, no. 6, Nov. 1981, pp. 487-501.

[22] Samet, H., "Region representation: quadtrees from binary arrays." Comput. Gr. Image Process., vol. 13, no. 1, May 1980, pp. 88-93.

[23] Samet, H., "Region representation:boundary codes from quadtrees." Commun. ACM, vol. 23, no. 3, Mar. 1980, pp. 171-179.

[24] Samet, H., "Region representation:quadtrees from boundary codes." Commun. ACM, vol. 23, no. 3, Mar. 1980, pp. 163-170.

[25] Samet, H., "Deletion in two-dimensional quad trees." Commun. ACM, vol. 23, no. 12, Dec. 1980, pp. 703-710.

[26] Samet, H., "Neighbor finding techniques for images represented by quadtrees." Comput. Gr. Image Process., vol. 1, no. 1, Jan. 1982, pp. 37-57.

[27] Samet, H., "Distance transform for images represented by quadtrees." IEEE Trans. Pattern Analysis Mach. Intelligence, vol. 4, no. 3, May 1982, pp. 298-303.

[28] Samet, H., "Algorithms for converting rasters to quadtrees to rasters." IEEE Trans. Pattern Analysis Mach. Intelligence, vol. PAMI-3, no. 1, Jan. 1981, pp. 93-95.

[29] Samet, H., "Computing geometric properties of images represented by linear quadtrees." IEEE Trans. Pattern Analysis Mach. Intelligence, vol. 7, no. 1, Jan. 1985.

[30] Samet, H., "On encoding boundaries with quadtrees." IEEE Trans. Pattern Analysis Mach. Intelligence, vol. 6, no. 3, May 1984, pp. 365-369.

[31] Samet, H., "Data structures for quadtree approximation and compression." Commun. ACM, vol 28. no. 9, Nov. 1985, pp. 973-993.

[32] Samet, H., "The quadtree and related hierarchical data structures." ACM Comput. Surveys, vol. 16, no. 2, Jun. 1984, pp. 187-260.

[33] Samet, H., "A top-down quadtree traversal Algorithm." IEEE Trans. Pattern Analysis Mach. Intelligence, vol. PAMI-7, no. 1, Jan. 1985, pp. 94-97.

[34] Samet, H., "Computing perimeters of images represented by quadtrees." IEEE Trans. Pattern Analysis Mach. Intelligence, vol. 3, no. 6, Nov. 1981, pp. 683-687.

[35] Scott, D. S., and Iyengar, S. S., "TID - a translation invariant data structure for storing images." Commun. ACM, vol. 29, no. 5, Dec. 1986, pp. 418-429.

[36] Shneier, M., "Calculations of geometric properties using quadtrees." Comput. Gr. Image Process., vol. 16, no. 3, Jul. 1981, pp. 296-302.

[37] Shu-xiang, L., "Adjancy detection using quadcodes." Commun. ACM, vol. 30, no. 7, Jul. 1987, pp. 627-632.

[38] Shu-xiang, L., "The quadcode and its arithmetic." Commun. ACM, vol. 30, no. 7, Jul. 1987, pp. 621-626.

[39] Tamminen, M., "Encoding pixel trees." Comput. Vision, Gr. and Image Process., vol. 28, Jun. 1984, pp. 44-57.

[40] Wiseman, N. E., and Oliver, M. A., "Operations on quadtree encoded images." The Comput. J., vol. 26, no. 1, May 1983, pp. 83-93.

[41] Wiseman, N. E., and Oliver, M. A., "Operations on quadtree leaves and related image areas." The Comput. J., vol.26, no. 4, Mar. 1983, pp.375-380.

APPENDIXES

APPENDIX A

PROCEDURES AND FUNCTIONS

Set Operations

Intersection

This procedure finds the logical AND of the two binary images represented by quadtrees. Input to the procedure is a pointer to the root of each quadtree [36].

```
quadtree procedure INTERSECTION(TRRE1,TREE2)
     /* returns the intersection of TREE1 and TREE2 */
begin
     node TREE1,TREE2,INTERSECT;
     quadrant I;
     if BLACK(TREE1) or WHITE(TREE2) then
          return(COPY(TREE2));
     else
          if BLACK(TREE2) or WHITE(TREE1) then
               return(COPY(TREE1));
     INTERSECT:=CREATENODE(); /* create a root node */
     for I in {NW<NE<SW<SE} do
         begin
           SON(INTERSECT,I):=INTERSECTION(SON(TREE1,I);
                                          SON(TREE2,I));
           FATHER(SON(INTERSECT,I):=INTERSECT;
         end;
     return(INTERSECT);
end;
```

Union

Union of two quadtrees. Input to the procedure is a pointer to the root of each quadtree [36].

```
quadtree procedure UNION(TRRE1,TREE2)

     /* returns the union of TREE1 and TREE2 */
begin
     node TREE1,TREE2,UNI;
     quadrant I;
     if BLACK(TREE1) or WHITE(TREE2) then
          return(COPY(TREE2));
     else
          if BLACK(TREE2) or WHITE(TREE1) then
               return(COPY(TREE1));
     UNI:=CREATENODE(); /* create a root node */
     for I in {NW<NE<SW<SE} do
        begin
          SON(UNI,I):=UNION(SON(TREE1,I),SON(TREE2,I));
          FATHER(SON(UNI,I):=UNI;
        end;
     return(UNI);
end;
```

```
quadtree procedure COPY(TREE);
     /* creates a tree structure identical to TREE */
begin
     quadtree TREE, NEWTREE;
     quadrant I;
     NEWTREE:=CREATENODE();
          /* create a node with NULL FATHER, SON,and
                                          TYPE nodes*/
     TYPE(NEWTREE):=TYPE(TREE);
     for I in {NW,NE,SW,SE} do
          if SON(TRRE,I)=NULL then
               SON(NEWTREE,I):=NULL;
          else begin
               SON(NEWTREE,I):=COPY(SON(TREE,I));
               FATHER(SON(NEWTREE,I)):=NEWTREE;
          end;
     return(NEWTREE);
end;
```

Complement

Complement of a given quadtree (i.e., changing BLACK nodes into WHITE, and WHITE nodes into BLACK). Input to the procedure is a pointer to the root of the quadtree [36].

```
procedure COMPLEMENT(QUADTREE);
     /* change a quadtree into its complement */
begin
     node QUADTREE;
     quadrant I;
     if GRAY(QUADTREE) then
          for I in {NW,NE,SW,SE} do
               COMPLEMENT(QUADTREE,I);
     else
          if BLACK(QUADTREE) then
               TYPE(QUADTREE):=WHITE;
          else /* WHITE node */
               TYPE(QUADTREE):=BLACK;
end;
```

Geometric Transformation

Rotation

The following procedure [8] rotates the image in the quadtree by 90 degrees in the clockwise direction.

```
procedure ROTATE(var node: node_ptr);
var
     old-NWest, old_NEast: node_ptr;
     old_SWest, old_SWest: node_ptr;
begin
     Old_NWest := node^.NWest;
     Old_NEest := node^.NEest;
     Old_SWest := node^.SWest;
     Old_SEest := node^.SEest;

     node^.NWest := old_SWest;
     node^.NEest := old_NWest;
     node^.SWest := old_SEest;
     node^.SEest := old_NEest;

     if (node^.NWest <> Nil) then ROTATE (nod^.NWest);
     if (node^.NEest <> Nil) then ROTATE (nod^.NEest);
     if (node^.SWest <> Nil) then ROTATE (nod^.SWest);
     if (node^.SEest <> Nil) then ROTATE (nod^.SEest);
end;
```

Area

This procedure [36] finds the area of an image represented by a quadtree.

```
integer procedure AREA(QUADTREE,N);
   begin
     node QUADTREE;
     integer BLACKAREA;
     level N;
     quadrant I;
     BLACKAREA:=0;
     if GRAY(QUADTREE) then
       for I in {NW,NE,SW,SE} do
         BLACKAREA:=BLACKAREA+AREA(SON(QUADTREE,I),N1);
       else if BLACK(QUADTREE) then
         BLACKAREA:=B;LACKAREA+2^(2*N);
     return(BLACKAREA);
   end;
```

Procedures and Functions Developed During Software Development

Constructing a Complete Quadtree

This function constructs a "l" levels complete quadtree. The root is at level "l-1" and the leaf nodes at level zero. The function returns a pointer to the root.

```
function build_quadtree(l:integer):node_ptr;
var
   node:node_ptr;

begin

     new(node);
     if l=0 then
          (* if at leaf nodes level *)

        begin
             node^.color:=black;
             node^.nw:=nil;
             node^.ne:=nil;
             node^.sw:=nil;
             node^.se:=nil;
        end

     else
          (* internal node level *)
        begin
            node^.color:=gray;
            node^.nw:=build_quadtree(l-1);
            node^.color:=gray;
            node^.ne:=build_quadtree(l-1);
            node^.color:=gray;
            node^.sw:=build_quadtree(l-1);
            node^.color:=gray;
            node^.se:=build_quadtree(l-1);
       end;

  build_quadtree:=node;

end;(* build_quadtree*)
```

Postorder Traversal of a Quadtree

This procedure performs a postorder traversal of a given quadtree (i.e., NW,NE,SW,SE). In addition, the procedure prints the quadtree nodes as follows: "B", for BLACK node, "W" for WHITE node, and "G" for GRAY node (i.e., internal node)

```
procedure post_order(node:node_ptr);
     (* node - root of the quadtree *)

begin

     if node^.color=gray then
          (* internal node *)

        begin
             write('G ');
             post_order(node^.nw);
             post_order(node^.ne);
             post_order(node^.sw);
             post_order(node^.se);
        end

     else
          (* leaf node *)

         begin
              if node^.color = black then
                 write('B ');
              if node^.color = white then
                 write('W ');
         end;

end; (* post_order *)
```

Plot Quadtree

This procedure displays the image represented by a given quadtree. Three parameters are passed to the procedure:

. the root of the quadtree

. the coordinate where the upper left corner of the image is to be displayed on the screen

. the size of the image (size is in powers of two)

```
procedure plot_quadtree(node:node_ptr;
                        x1,y1,length:integer);

var
     x2,y2,hlength:integer;

begin

     hlength:=length div 2;
     x2:=x1+hlength;
     y2:=y1+hlength;

     if node^.color=gray then
          (* internal node *)

        begin
             plot_quadtree(node^.nw,x1,y1,hlength);
             plot_quadtree(node^.ne,x1,y2,hlength);
             plot_quadtree(node^.sw,x2,y1,hlength);
             plot_quadtree(node^.se,x2,y2,hlength);
        end

     else
          (* leaf node *)

          if node^.color=black then
               bar(x1,y1,x1+length,y1+length);

end; (* plot_quadtree *)
```

Create an Index File

This procedure generates the scanning coordinates explained in Chapter IV. Three parameters are passed to the procedure:

. pointer to the root of a complete quadtree

. number of levels in the complete quadtree

. coordinate (x,y), were x=1 and y=1

```
procedure create_indexfile(node:node_ptr;
                           var indexfile:text;
                           l,x,y:integer);

(*  power(n) returns 2^n  *)

begin

     if l=1 then
         (* if the node is a leafnode parent *)

         writeln(indexfile,x,' ',y);

     if (node^.nw <> nil) and (l >1) then
          create_indexfile(node^.nw,indexfile,
                           l-1,x,y);

     if (node^.ne <> nil) and (l>1) then
          create_indexfile(node^.ne,indexfile,l-1,
                           x,y+power(l-1));

     if (node^.sw <> nil) and (l>1) then
          create_indexfile(node^.sw,indexfile,l-1,
                           x+power(l-1),y);

     if (node^.se <> nil) and (l>1) then
          create_indexfile(node^.se,indexfile,l-1,
                           x+power(l-1),y+power(l-1));

end;(* create_indexfile *)
```

Scanning an Image and "Coloring" Its Quadtree

This procedure scans an imaged displayed on the screen and "colors" the leaf nodes of the given complete quadtree as explained in Chapter IV. Two parameters are passed to the procedure:

. pointer to the complete quadtree root

. index file name containing the scanning coordinates.

```
procedure scan_image_on_screen(node:node_ptr;
                               var  indexf:text);

var
     x,y:integer;
begin
     if node^.nw^.nw <> nil then
          scan_image_on_screen(node^.nw,indexf);
     if node^.ne^.ne <> nil then
          scan_image_on_screen(node^.ne,indexf);
     if node^.sw^.sw <> nil then
          scan_image_on_screen(node^.sw,indexf);
     if node^.se^.se <> nil then
          scan_image_on_screen(node^.se,indexf);
     if node^.nw^.nw = nil then
        begin
             readln(indexf,x,y);
             if getpixel(x,y)=0 then
               node^.nw^.color:=white
             else
               node^.nw^.color:=black;
             if getpixel(x,y+1)=0 then
               node^.ne^.color:=white
             else
               node^.ne^.color:=black;
             if getpixel(x+1,y)=0 then
               node^.sw^.color:=white
             else
               node^.sw^.color:=black;
             if getpixel(x+1,y+1)=0then
               node^.se^.color:=white
             else
               node^.se^.color:=black;
 end;
end;(* scan_image_on_screen *)
```

Saving Quadtree Using DF-expression Method

This procedure saves a given quadtree in a text file using the DF-expression method described in Chapter II. Two parameters are passed to the procedure:

. pointer to the quadtree root

. file name

```
procedure save_quadimage(node:node_ptr;
                         var imagefile:text);

begin

     if node^.color = gray then
          (* internal node *)

        begin
             writeln(imagefile,'(');
             save_quadimage(node^.nw,imagefile);
             save_quadimage(node^.ne,imagefile);
             save_quadimage(node^.sw,imagefile);
             save_quadimage(node^.se,imagefile);
        end

     else
          (* leaf node)

         if node^.color = black then
            writeln(imagefile,'B')
         else
             writeln(imagefile,'W');

end;(* save_quadimage *)
```

Reconstructing a Quadtree from DF-expression Representation

This function reconstructs a quadtree from an image that originally was represented by a quadtree and was saved in a file using the DF-expression method. The function returns a pointer to the quadtree root.

```
function quadimage_from_quadfile(var imagefile:
                                        text):node_ptr;

var
     node:node_ptr;
     ch:char;

begin

     new(node);
     readln(imagefile,ch);
     if ch = '(' then
          (* internal node *)

        begin
          node^.color:=gray;
          node^.nw:=quadimage_from_quadfile(imagefile);
          node^.ne:=quadimage_from_quadfile(imagefile);
          node^.sw:=quadimage_from_quadfile(imagefile);
          node^.se:=quadimage_from_quadfile(imagefile);
        end

    else
          (* leaf node *)

        begin
            if ch = 'B' then
                 node^.color:=black
            else
                 node^.color:=white;
            node^.nw:=nil;
            node^.ne:=nil;
            node^.sw:=nil;
            node^.se:=nil;
         end;

    quadimage_from_quadfile:=node;

end;(* quadimage_from_quadfile *)
```

APPENDIX B

TABLES AND FIGURES

In this appendix, two tables are given and various screens, generated by the software, are also included.

For each consonant and vowel quadtree, the tables include the following data:

. the number of leaf nodes in the quadtree

. the total number of nodes in the quadtree

Note that these quadtrees are the result of the merging process described in Chapter IV (see implementation steps section). The original complete quadtree had 1365 nodes! (for a region of size 2^5 x 2^5, 1024 leaf nodes and 341 interanl nodes).

Note also, that actually, only three vowel quadtrees were constructed: the vowels KAMATZ, PATACH, and HIRIK. The remaining vowels were displayed using combinations of the three vowels as follows:

. the vowel TZAREH is the combination of two HIRIKs

. the vowel SEGOL is the combination of three HIRIKs

. the vowel SHEVAH is the combination of two HIRIKs

- the vowel KUBUTZ is the combination of three HIRIKs
- the vowel CHOLAM is the combination of the consonant VAV and the vowel HIRIK
- the vowel SHURUK is the combination of the consonant VAV and the vowel HIRIK

The screens included are the consonants and vowels of the Hebrew language, with a few examples of Hebrew words.

TABLE I

NUMBER OF LEAF NODES AND NUMBER OF TOTAL NODES
OF EACH CONSONANT QUADTREE

Consonant Name	Number of Leaf nodes	Number of Total nodes
ALEF	196	261
BET	55	73
GIMEL	115	153
DALET	82	109
HAY	82	109
VAV	67	89
ZAIN	136	181
HET	64	85
TET	88	117
YOD	64	85
KAF	73	97
FINAL KAF	70	93
LAMED	115	153
MEM	124	165
FINAL MEM	82	109
NUN	64	85
FINAL NUN	67	89
SAMEH	148	197
AYIN	142	189
PAY	73	97
FINAL PAY	67	89
TZADEE	175	233
FINAL TZADEE	190	253
KOOF	127	169
RESH	64	85
SHIN	97	129
TAV	82	109

TABLE II

NUMBER OF LEAF NODES AND NUMBER OF TOTAL NODES
OF EACH VOWEL QUADTREE

Vowel Name	Number of Leaf nodes	Number of Total nodes
KAMATZ	256	341
PATACH	256	341
HIRIK	304	405

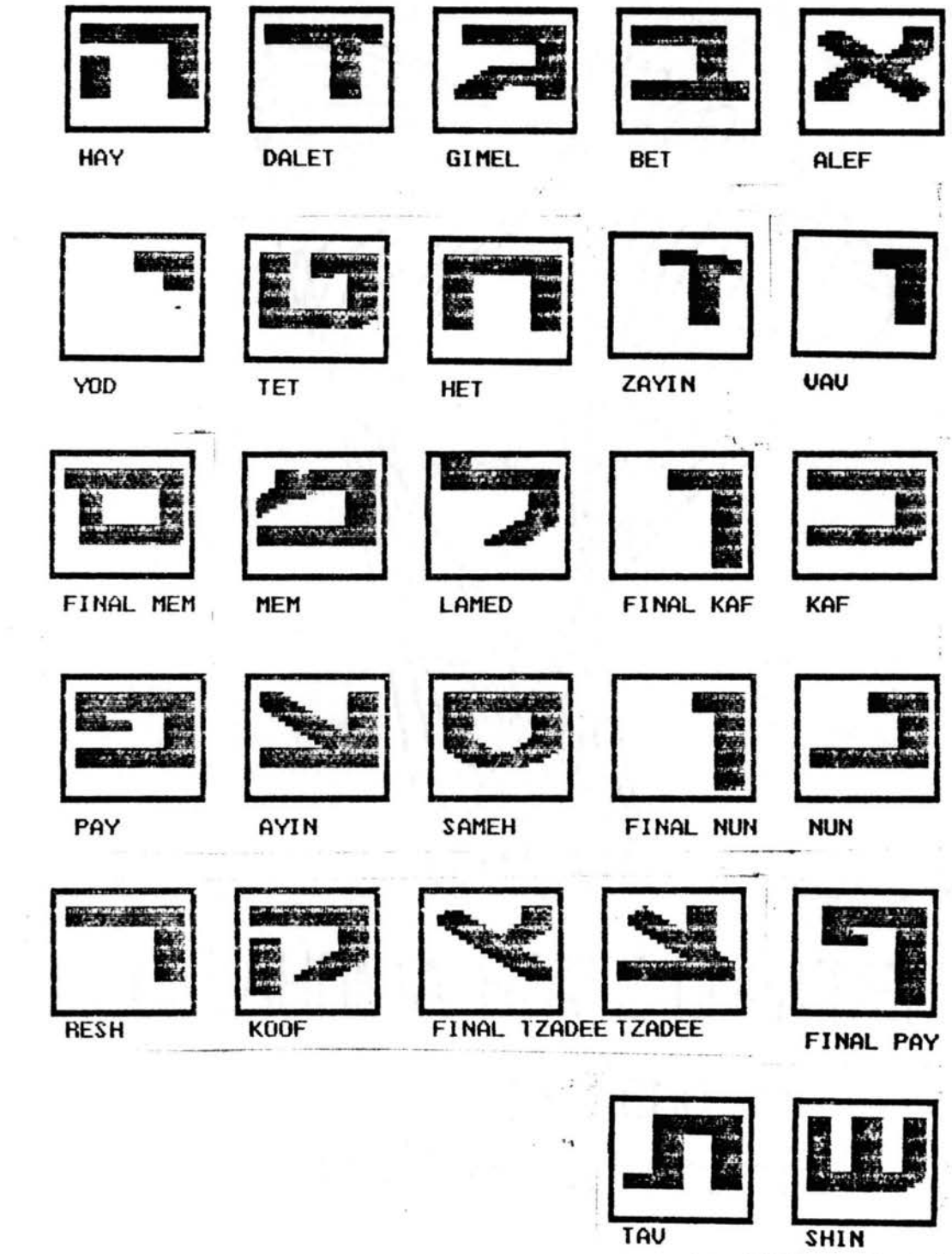


Figure 21. The Hebrew Consonants

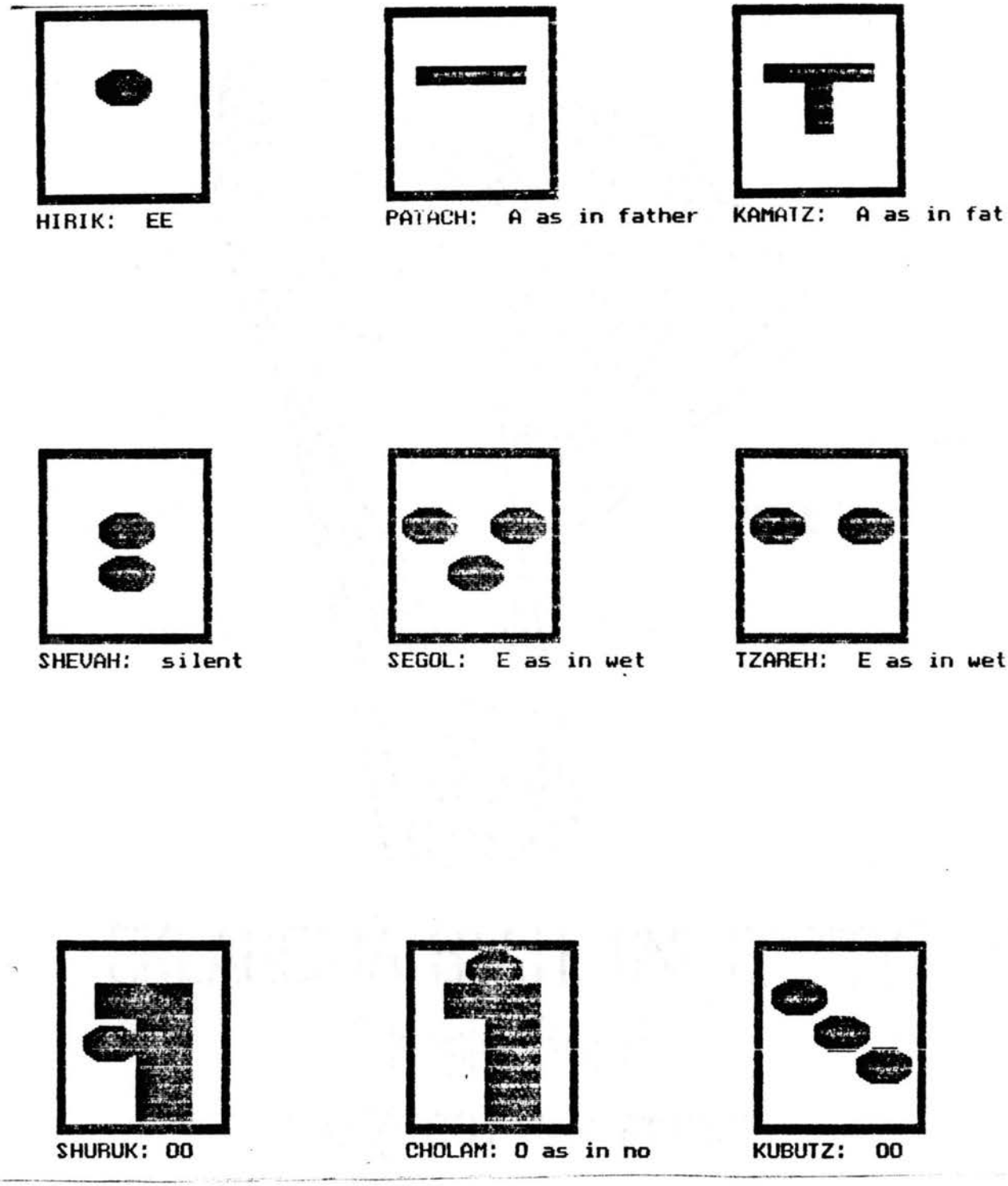


Figure 22. The Hebrew vowels

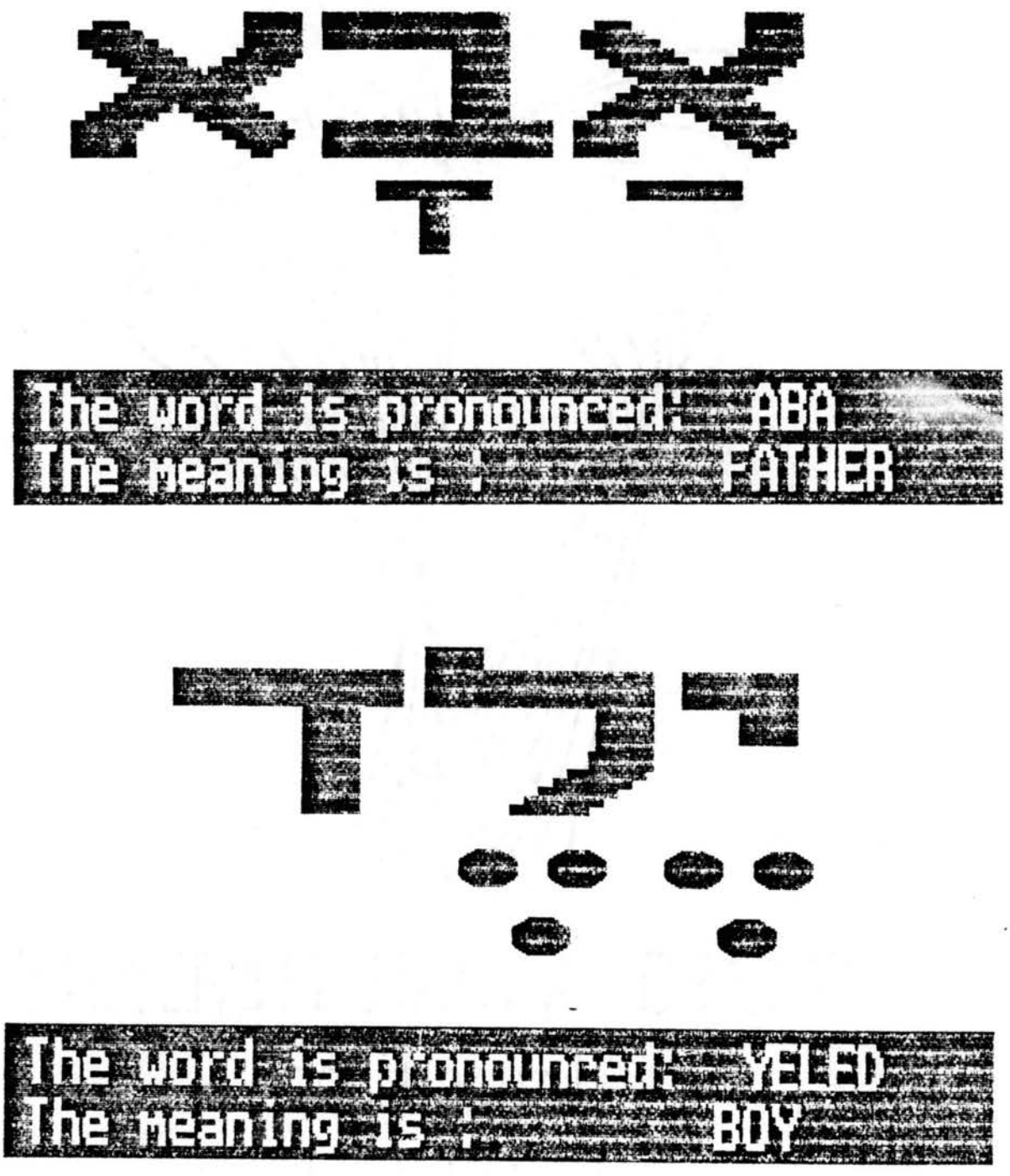
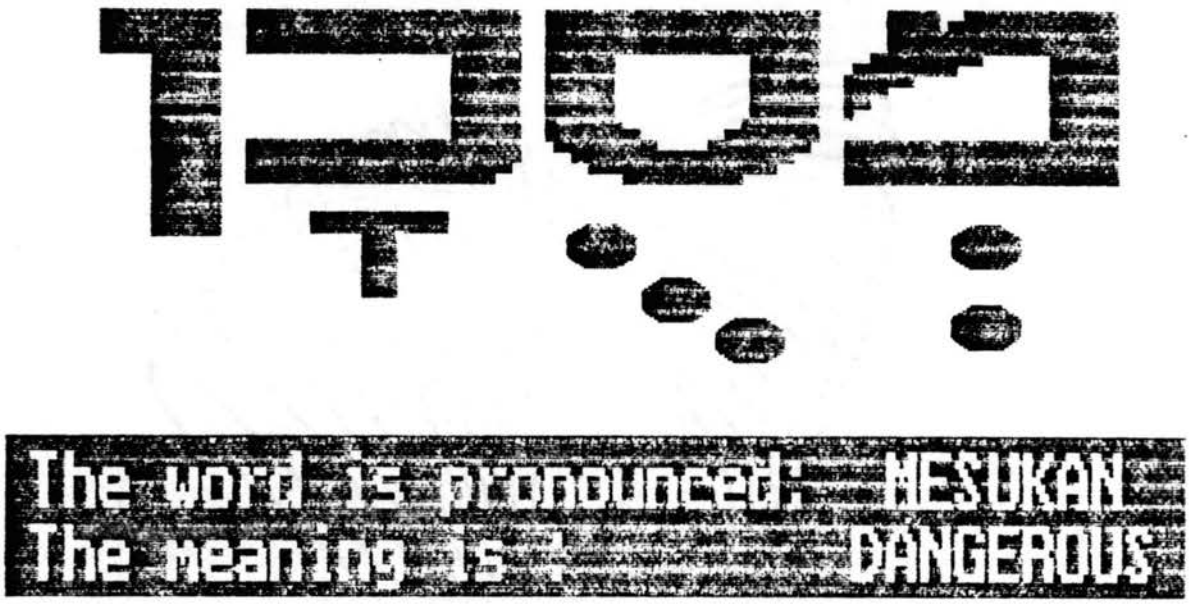


Figure 23. Examples of Hebrew words



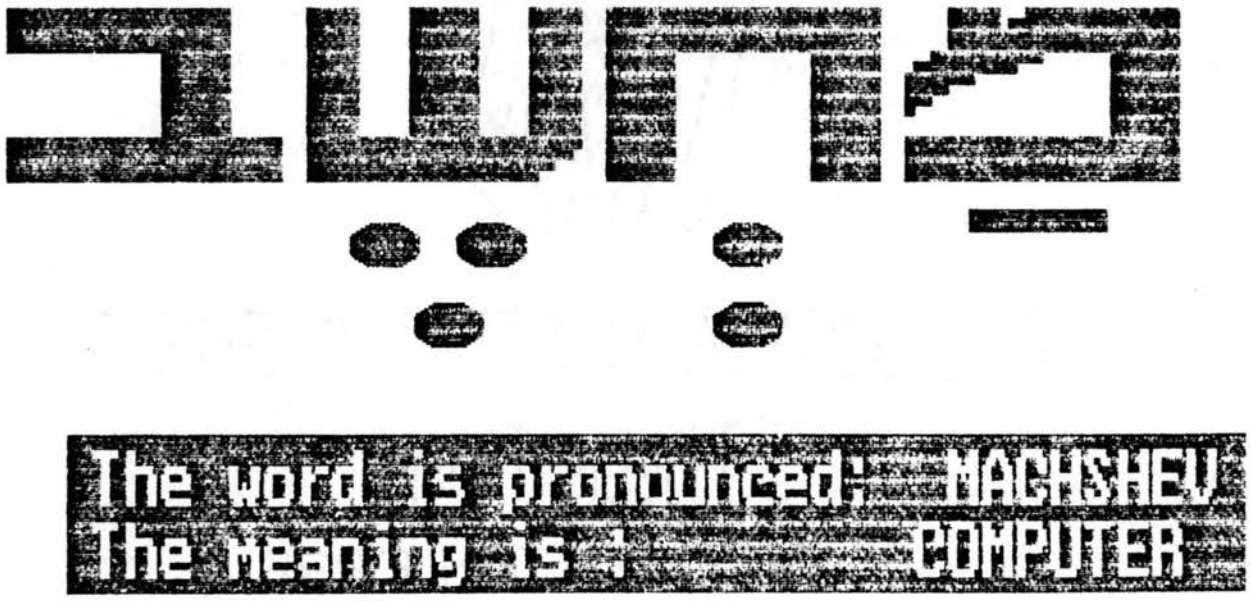
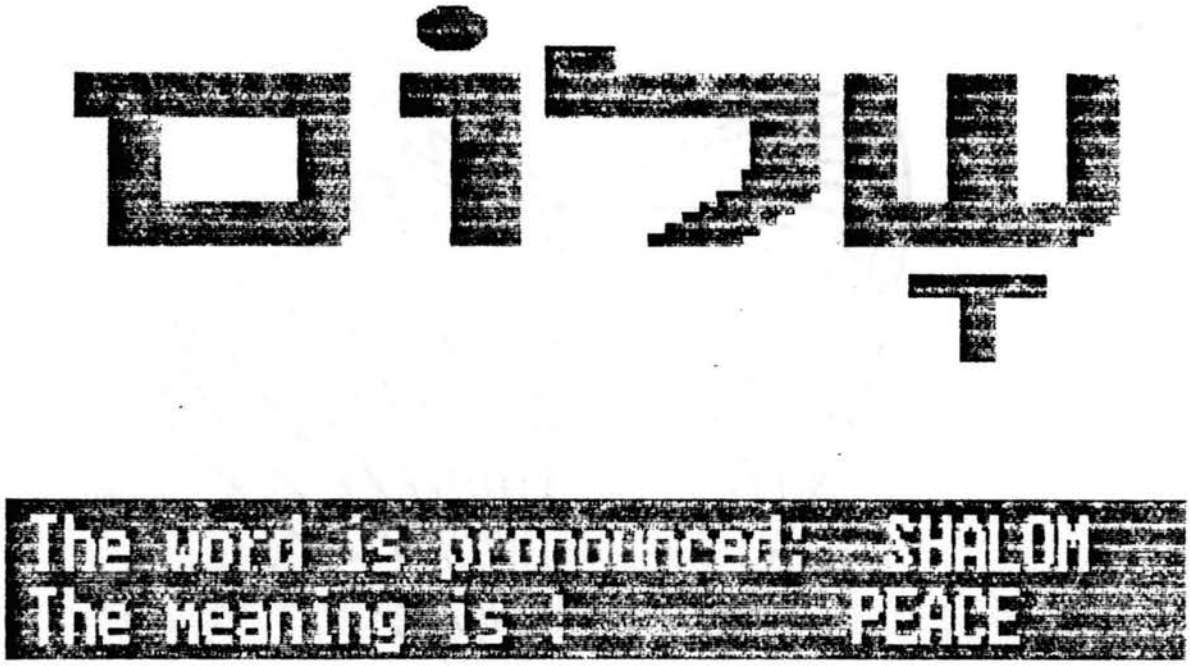


Figure 24. Examples of Hebrew words



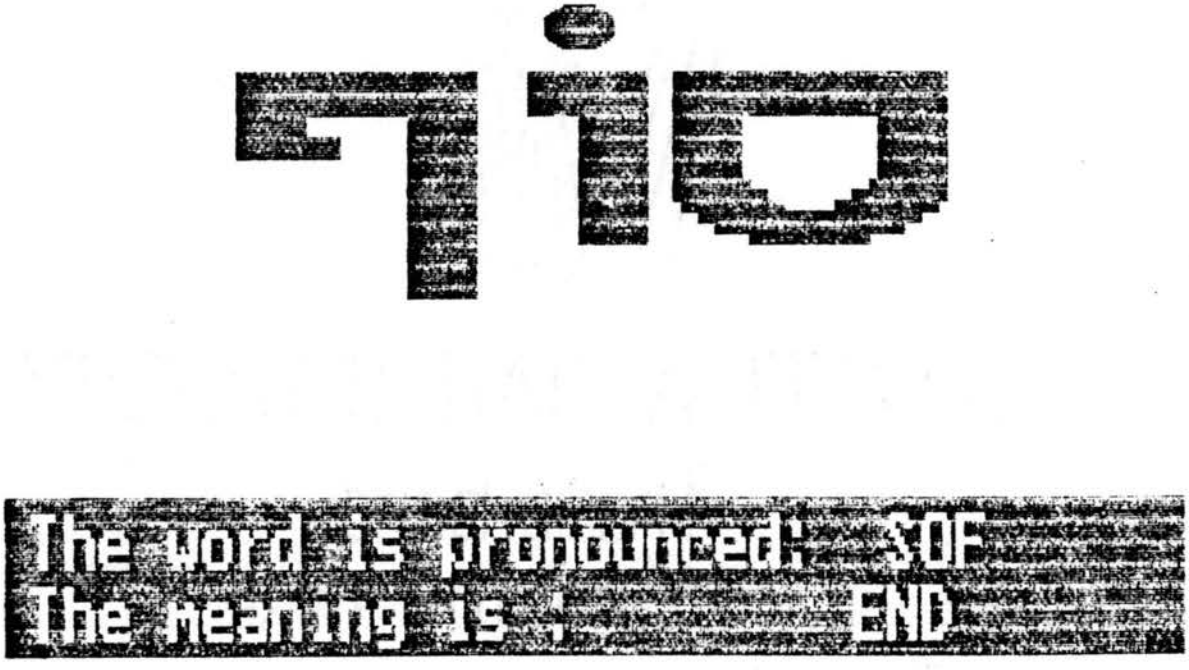


Figure 25. Examples of Hebrew words

VITA

Israel Shuval

Candidate for the Degree of

Master of Science

Thesis: REPRESENTING IMAGES USING THE QUADTREE DATA STRUCTURE (HEBREW CONSONANTS AND VOWELS)

Major Field: Computing and information Sciences

Biographical:

Personal Data: Born in Israel, September 29, 1957, the son of David and Margalit Shvili. Married to Alexandria Shuval on August 8, 1982.

Education: Graduate from Ort Givatime High school, Givatime, Israel, in July 1976; received Bachelor of Science degree in Computer Science from Oklahoma City University at Oklahoma City, in December, 1986; completed requirements for the Master of Science degree at Oklahoma State University in July, 1989.

Professional Experience: Programmer, Phillips Petroleum Company, January 1989 to present; Adjunct Professor, Department of Computer Science, Central State University, August 1987 to December 1988.