Pattern Recognition Vol. 17, No. 6, pp. 647–656, 1984.
Printed in Great Britain.

0031-3203/84 $3.00+.00
Pergamon Press Ltd.

A GEOGRAPHIC INFORMATION SYSTEM USING QUADTREES*

HANAN SAMET,† AZRIEL ROSENFELD, CLIFFORD A. SHAFFER and ROBERT E. WEBBER
Computer Science Department and Center for Automation Research, University of Maryland, College Park, MD 20742, U.S.A.

(*Received* 13 *December* 1983; *in revised form* 28 *February* 1984; *received for publication* 15 *March* 1984)

Abstract—We describe the current status of an ongoing research effort to develop a geographic information system based on quadtrees. Quadtree encodings were constructed for area, point and line features for a small area in Northern California. The encoding used was a variant of the linear quadtree. The implementation used a B-tree to organize the list of leaves and allow management of trees too large to fit in core memory. Several database query functions have been implemented, including set operations, region property computations, map editing functions and map subset and windowing functions. A user of the system may access the database via an English-like query language.

Quadtrees Geographic information systems Image processing Region representation Curve representation

1. INTRODUCTION

The quadtree representation of regions, first proposed by Klinger,[1] has been the subject of intensive research over the past several years (for an overview, see the survey by Samet[2]). Numerous algorithms have been developed for constructing compact quadtree representations, converting between them and other region representations, computing region properties from them and computing the quadtree representations of Boolean combinations of regions from those of the given regions. Quadtrees have traditionally been implemented as trees which require space for the pointers from a node to its sons. Recently, there has been a considerable amount of interest in pointer-less quadtree representations[3,4] termed *linear quadtrees*. In this case, the set of regions is treated as a collection of leaf nodes. Each leaf is represented by use of a locational code corresponding to a sequence of directional codes that locate the leaf along a path from the root of the tree.

In this paper we describe the current status of an ongoing research effort to develop a geographic information system based on quadtrees. Quadtree encodings were constructed for area, point and line features from maps and overlays representing a small area of Northern California. The encoding used was a variant of the linear quadtree. A memory management system based on B-trees[5] was devised to organize the resulting collection of leaf nodes, allowing for the use of arbitrary sized maps within a restricted amount of core memory. Many database functions were implemented, including map editing capabilities, set operations and region property functions. Further details about this effort can be found in two earlier papers.[6,7]

The system described in this paper is intended primarily to demonstrate the efficiency of quadtrees as data structures for handling geographical entities. It is a contribution to geographic information system (GIS) design only on the level of the underlying data structures. Existing GIS's have been primarily based on polygon data structures, which do not lend them-

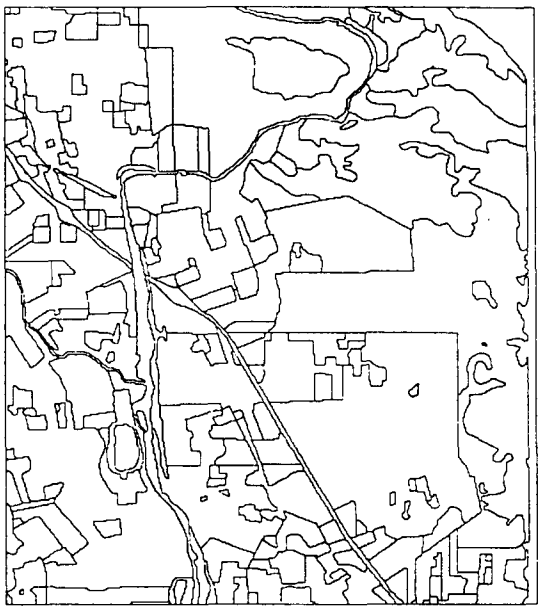

Fig. 1. The landuse map.

* The support of the U.S. Army Engineer Topographic Laboratories under Contract DAAK70-81-C-0059 is gratefully acknowledged.

† To whom correspondence should be addressed.

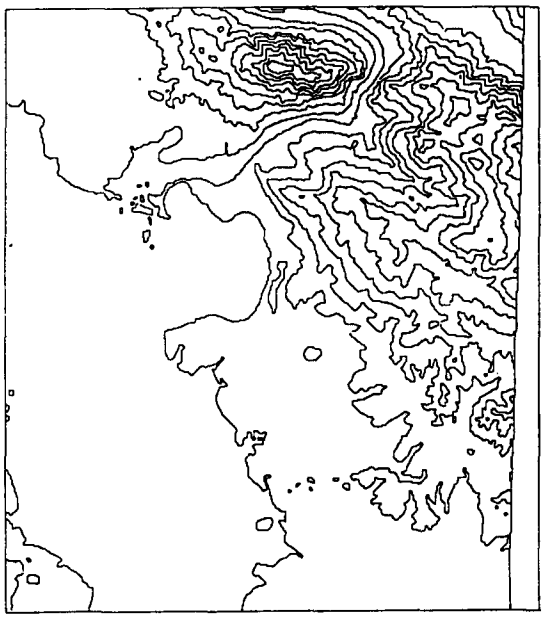

Fig. 2. The topography map.

Fig. 4. The house map.

selves as efficiently as quadtrees to the handling of many types of basic queries. Some recent examples of GIS's can be found in a journal special issue[8] and several individual papers.[9–12]

The database used in the study was supplied by the U.S. Army Engineer Topographic Laboratory, Ft. Belvoir, VA. The area data consisted of three registered map overlays representing landuse classes, terrain elevation contours and floodplain boundaries. The overlays were hand-digitized resulting in three arrays of size 400 × 450 pixels. Labels were associated with the pixels in each of the resulting regions, specifying the particular landuse class or elevation range. The regions were subsequently embedded within a 512 × 512 grid and quadtree encoded. The results are shown in Figs 1–3. We also made use of a geographic survey map for this area, from which we extracted point and line data. The house locations were digitized for a point map (Fig. 4), and four line maps were constructed corresponding to a railroad line, a power line, a city boundary and the road network from the area (Figs 5–8).

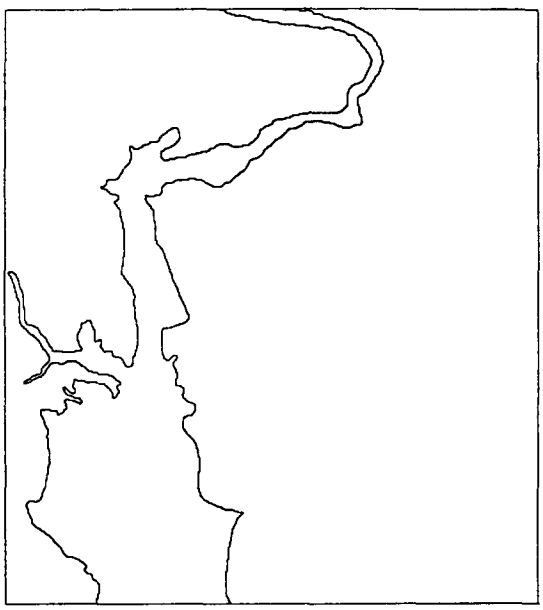

Fig. 3. The floodplain map.

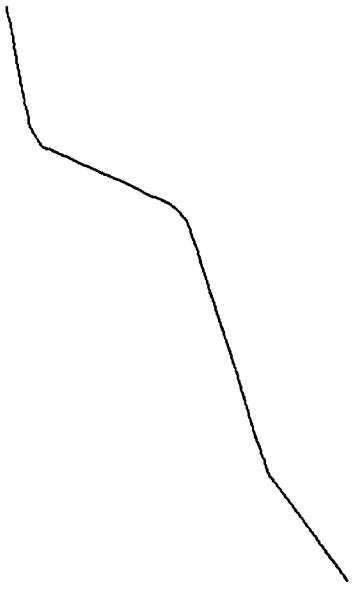

Fig. 5. The railroad map.

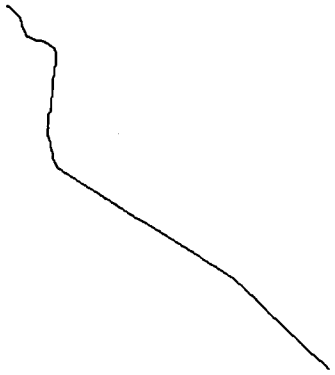

Fig. 6. The powerline map.

Fig. 8. The road map.

The rest of this paper is organized as follows. Section 2 describes the quadtree memory management system for storing and manipulating large quadtrees in external storage. Section 3 contains an outline of the quadtree editor which enables the interactive construction and updating of maps stored as quadtrees. Section 4 discusses the representations that we use for points and linear features. Section 5 gives an outline of the type of operations which we are currently able to perform on our database, while Section 6 describes the query language which we use to interact with the database.

2. THE QUADTREE MEMORY MANAGEMENT SYSTEM

Prior to discussing the memory management system, it is appropriate to briefly review the definition of a region quadtree. Given a $2^n \times 2^n$ array of pixels, a quadtree is constructed by repeatedly subdividing the array into quadrants, subquadrants, ... until we obtain blocks (possibly single pixels) which consist of a single value (e.g. a color). This process is represented by a tree of out degree four in which the root node corresponds to the entire array, the four sons of the root node

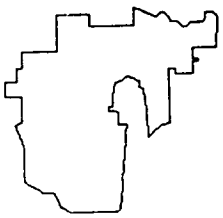

Fig. 7. The city border map.

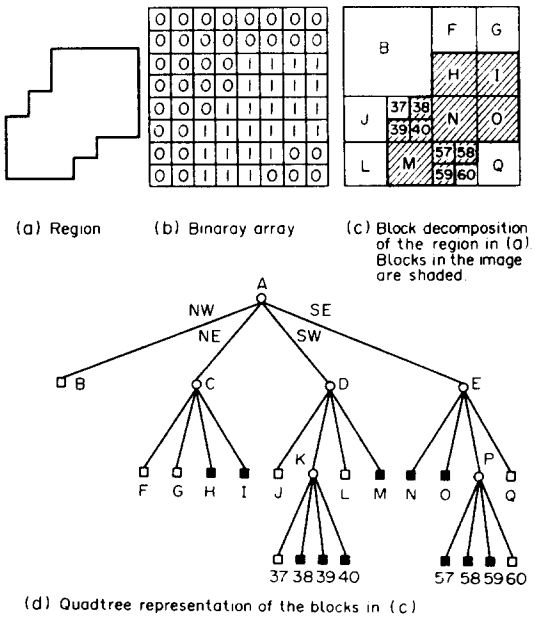

Fig. 9. A region, its binary array, its maximal blocks and the corresponding quadtree.

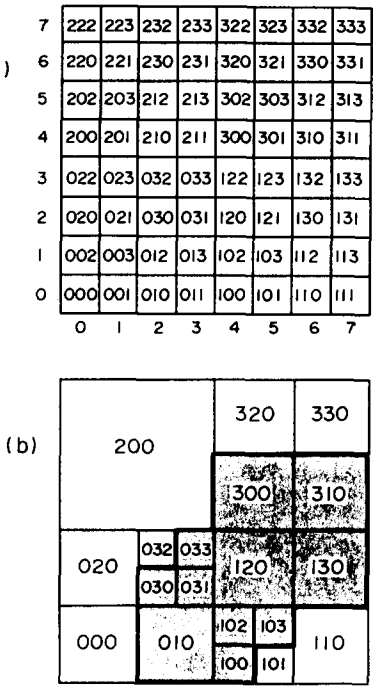

Fig. 10. An example demonstrating the use of locational codes to address blocks in an image represented by a quadtree.

correspond to the quadrants and the terminal nodes correspond to those blocks of the array for which no further subdivision is necessary. The nodes at level k (if any) represent blocks of size $2^k \times 2^k$ and are often referred to as nodes of size 2^k. Thus a node at level 0 corresponds to a single pixel in the image, while a node at level n is the root of the quadtree. For example, consider the region shown in Fig. 9a, which is represented by the $2^3 \times 2^3$ binary array in Fig. 9b. The resulting blocks for Fig. 9b are shown in Fig. 9c and the tree in Fig. 9d.

Note that the variant of the quadtree that we use results in a decomposition of space into equal-sized parts. This is in contrast to the point quadtree[13] and the k-d tree[14] where the decomposition is governed by the input. The advantage of our variant of the quadtree is that different maps will be in registration, thereby facilitating set operations such as map overlay.

Our database system can be viewed as being made up of four levels. The lowest level (henceforth known as the kernel) controls the interface between the disk file used to store the quadtree data and the programs that are used to manipulate the images. This level was written in the C programming language.[15] The next level, also written in C, contains the programs which implement the database algorithms—e.g. editing, set functions, etc. This level accesses the kernel (and hence the file system) strictly through a set of primitive functions. These functions allow the programmer of the second level to view the quadtree as an abstract data type, without worrying about the implementation details. The third level, written in LISP, allows convenient access to and composition of the database functions implemented by the second level. LISP is well suited to the manipulation of symbolic information. It is used here to keep track of the maps known by the database and information about landuse classes or elevation levels. User defined names and data items are maintained at this level. The highest level is the English-like query language described in Section 6.

Our implementation of the kernel stems from the linear quadtree encoding scheme used. In this scheme, each pixel is given an address derived from interleaving the bits of the binary representation of the pixel coordinates. Figure 10a shows a $2^3 \times 2^3$ grid with each pixel labelled in this fashion. Figure 10b shows the block decomposition of the region from Fig. 9a, with each square of the decomposition labelled with the address of its lower left pixel. The addresses in Fig. 9 are shown in base four, i.e. two bits are used for each digit. These addresses can also be viewed as a sequence of directional codes leading from the root of the tree to the node being described.

The key feature of our encoding scheme is that a preorder traversal of the explicit tree will produce the nodes in ascending order of the addresses of the pixels at their lower left corner. We store only the leaf nodes of the tree, sorted in ascending order of this address field. Any pixel contained within a node will have an address greater than that of the leaf's lower left corner, but less than that of the next node in preorder. Therefore, given the address of any pixel and a list of leaves ordered by their addresses, finding the leaf containing that pixel reduces to searching a sorted list.

Given the linear ordering of leaf nodes and the fact that we are storing files containing as many as 30,000–40,000 leaves, we decided to organize the quadtree files using a B-tree structure. The kernel is primarily a collection of routines that maintain a buffer pool in core and a B-tree in the disk file. The buffer pool need only store that portion of the tree in core for which there is room. We expect that there will be strong locality of reference—i.e. the leaf for which we are presently searching will very likely be near the leaf we last found. Therefore, the buffer pool is maintained on a schedule that replaces the least recently used buffers first.

At the level of the kernel, the three types of quadtrees (for point, line and region data) are identical. A quadtree node descriptor is composed of two 32-bit words. The first word contains information on the leaf's position and depth in the tree. In particular, it contains a 24 bit field which consists of the address of the lower left corner of the node formed by interleaving the bits of its x and y coordinates. The remaining eight bits indicate the depth of the node in the tree. The second word contains information about the data that the node represents. The contents of the second word are not used by the kernel.

A quadtree file is made up of four parts. First, there is the kernel's fixed-size header, which contains infor-

mation about the size of the file and the B-tree structure. Second, there is a fixed-size block for the user's header (further described in Section 3). Typical usages of the user's header would be to keep track of whether a quadtree file is to be interpreted as a point quadtree or a region quadtree, the x and y coordinates of the lower-left-hand corner of the quadtree (with respect to some global coordinate system) and the width of the quadtree in pixels. The third part is a list of comments. These comments are either generated by database functions when a new map is created or are inserted at the request of the database user. In either case, they serve to document a map. Finally, a quadtree file contains the list of B-tree pages that contain the quadtree nodes.

The bulk of the quadtree file is made up of the list of B-tree pages. Each page is 512 bytes long (an optimal size for the programming language's read and write routines). We store 60 quadtree leaves in each page. The remaining space in the page contains information related to the B-tree organization of the database. When a leaf is requested, the page containing it is read into the buffer pool, possibly displacing another previously used page.

3. THE QUADTREE EDITOR

The quadtree editor exists to facilitate the interactive construction and updating of maps stored as quadtrees. Presently, it is a separate program from the database system, written entirely in C, with its own command language. Rather than forcing the user to think in terms of the tree structure, the editor's tree manipulation commands make references to logical units of the map (e.g. lines, points or polygons). The user can perform editing operations such as inserting a line or point, changing the value of a specified polygon or splitting a specified polygon into more than one piece.

When many changes are to be made, the user may wish to see the effects of each step. Commands are provided to enable him to examine all or part of the map at a selected location on a display device. This display is continuously updated as further map manipulation commands are executed. Associated with each map's quadtree representation is a descriptor termed the quadtree header. There exist commands which allow the user to modify this header. The header contains the size of the map, the tree type (area, point or line), the coordinate of the lower left corner of the map in relation to a global coordinate system, the rotation angle or tilt of the map from the external horizontal and some information as to the type of data (i.e. topography, landuse, house) that is being stored. A command is also provided to enable the user to insert textual comments for documentation purposes.

In order for the quadtree editor to be useful, a set of map manipulating functions is needed that permits the user to create any desired map. The user of a geographic information system such as ours views the units of his map in terms of logical units, such as 'lines' or 'polygons', and not square 'nodes'. Therefore, for region maps it is clear that the most natural implementation is one that permits the modifying commands to make changes to specified polygons. This means that the implementation of these commands must enable modification of all the nodes which make up a polygon or group of polygons without affecting nodes of neighboring polygons. We view each polygon (and hence each node making up the polygon) as a member of a 'class'. This class could be an elevation range or a landuse type, such as 'wheatfields'. Class information is recorded for each node by use of a value field that is part of the node's descriptor.

The editor is an interpretive system—i.e. the user gives a command and it is executed, after which the system is ready to execute the next command. There is no notion of composing functions, as there is in the quadtree database language. Area maps are updated by use of the REPLACE, CHANGE and SPLIT commands, which replace all polygons of a given value with a new value, change the value of a given polygon or split a polygon into multiple polygons, respectively. Line and point maps are updated by use of the INSERT and DELETE commands, which insert or delete lines or points, respectively. In order that the user may see what he is editing, there are commands that draw all or part of the map onto a selected section of a display device. The user may also alter the header of the map.

When the editor is invoked, the user gives the name of the file to be edited. A temporary disk file is created on which all editing is to be done. Another file is created to store the commands given by the user. These files help protect the user from serious loss due to system crashes or his own errors, such as mistyped or unwanted commands. They also enable him to abort the editing session without damaging the original copy. If the file to be edited is an old one, a copy is made in the temporary file. If a new map is to be created, then a default header is installed and the map is initialized to be one WHITE region.

Changes to region maps are made by use of the REPLACE, CHANGE and SPLIT commands, as described below. Section 4 discusses the modification of line and point maps. The REPLACE command is executed by traversing the entire quadtree. Those nodes with the old (class) value have that value replaced by the new. For this command it is not necessary to distinguish between polygons of the same class, since they are all processed in the same way.

The CHANGE command is more complicated. It should manipulate only one polygon; however, other polygons of that class may also exist. After the command line has been parsed, a recursive function is called which actually performs the desired work. This function takes a node as its parameter. This node is checked to see that it has the old value (the one to be

changed). If so, then its value is changed to the new one and the function is recursively applied to all of the node's neighbors. In this way, all nodes of a polygon will eventually be reached and only nodes in the polygon will have their values changed (since only four-neighbors of nodes in the desired polygon are ever candidates for processing).

The SPLIT command allows the user to impose an arbitrary line, one pixel wide, of a designated value onto the map. It assumes that the arbitrary line is specified as a chain code. The intended use of the command is to split a polygon into two or more separate parts. One of these parts would then become a polygon of the same class as the pixels representing the arbitrary line via subsequent invocation of the CHANGE command. The pixels representing the arbitrary line would then be part of this new polygon. Alternatively, the SPLIT commands can be used to make slight modifications of only a very few pixels, such as correcting a slightly misplaced border of a polygon. This type of correction could not be applied in any other way with the available command set.

The SPLIT command operates by first inserting a one pixel node into the tree corresponding to the first coordinate given and then following the chaincode inserting nodes as it reads the code. As the user types the code, the code is also inserted into the command file. Allowing the user to observe the progress of the chaincode as he is inputting it is a key feature of our implementation of the SPLIT command. Typing an incorrect chaincode was judged to be a very common source of error when the implementation was designed. Enabling the user to see the line displayed as he inputs it allows the rapid detection and correction of errors. When the backspace key is typed, the chaincode is backed up one pixel. This is accomplished by examining the end of the command file. The last direction of the code is read to determine the coordinates of the previous pixel of the chaincode and then both the map and the command file are updated to reflect the backup.

By repeated use of the three commands REPLACE, CHANGE and SPLIT, it is possible to make any desired changes to a region map. Clearly this is true, since in the worst case the user could construct an entire map from one pixel chaincodes.

4. POINT AND LINE REPRESENTATIONS

Quadtree representations for point and line data were also developed. It should be noted that the same kernel (described in Section 2) is used for manipulating quadtrees of all three data types. When storing area data in region quadtrees, the value of a leaf corresponds to the color of the region that contains the leaf. Since there is no notion of color associated with either point or line data, other interpretations are placed on the information stored in the value portion of the leaf descriptor. The interpretation that a particular routine makes of a leaf's value is dependent on the type of data that is being stored in the quadtree. The database system stores the data type of the map in the user header, which was described in Section 3.

The value field of a quadtree node is made up of a single word 32 bits long. For area quadtrees there is simply a numeric value which can be interpreted as BLACK/WHITE, a color value, or as representing a symbolic item, such as landuse or elevation classes. In this last case, further information describing the item might be part of the database.

For point data, nodes containing data points are interpreted as containing the x coordinate (in the upper half of the word) and the y coordinate (in the lower half of the word). This coordinate pair is always in relation to the lower left corner of the map. If the coordinate of the data point in relation to a global coordinate system is desired, we simply add this offset to the coordinate of the lower left corner of the map, obtainable from the quadtree header. A single word 32 bits long is sufficient to describe a coordinate point, as the kernel limits the tree to a depth of twelve (i.e. a 4096×4096 pixel image). Nodes that do not contain a data point are represented by the value WHITE.

The above interpretation of the value field of a leaf descriptor has the following consequences with respect to quadtree algorithms that handle point data. No more than one data point can be stored in a quadtree leaf. Insertion of a point in a point data quadtree works as follows. First, we find the leaf that contains the point's location. If the leaf is empty, then the point's x and y coordinates are entered in the leaf's descriptor. Otherwise, the leaf is split into its four sons, the old leaf's point value is copied into the appropriate son and insertion is re-attempted. Deletion of a point in a point data quadtree is a matter of finding the leaf that contains the point and then changing the leaf's descriptor to that of an empty leaf. Next, we check to see if it is possible to merge the new empty leaf with its siblings.

The point data quadtree described above is termed a PR quadtree[2] and is also described by Orenstein.[16] It differs from the original point quadtree of Finkel and Bentley[13] in that the structure of the PR quadtree is independent of the order of point insertion. This is a result of the fact that leaves are always split by the kernel into four congruent squares (conforming to an area quadtree decomposition). In contrast, the splitting points for the point quadtree are the data points themselves, thereby resulting in four rectangles that are not necessarily equal in size.

When implementing line data, we decided that it would be desirable to be able to use the same kernel software as for area and point trees. This means that the node value field remains limited to 32 bits. We developed a variant of the edge quadtree of Shneier[17] which can cope with these limitations. We use the term *edge quadtree* in our discussion. Non-WHITE edge nodes contain exactly one line segment which inter-

sects two of the node's edges. When inserting a new line into the tree, nodes which would not conform to this requirement are quartered and re-inserted as appropriate. There are two special cases which must be handled. First, when two or more lines intersect, the point of intersection will never contain only one line segment. Special consideration must therefore be made for single pixel nodes (in this case the intersect point). Second, if a line only passes through one edge of a leaf node (i.e. it does not join with another edge), then we subdivide the node. In the worst case, the node may need to be subdivided to the single pixel level.

The value field of the edge quadtree leaf descriptor has four subfields. The first subfield (one bit) indicates error values. The second subfield (one bit) is off if the node is either WHITE or contains a single pixel. The bit is on if the node contains a line segment. The third subfield (two bits) for all non-WHITE nodes tells which son a node is with respect to its father. By setting this field, we guarantee that the leaf will not be automatically merged with its brothers by the kernel's insert routine. As this field is not set when the leaf contains no line segments, four empty quadrants are automatically merged together.

The fourth subfield (28 bits) of the value field of the edge quadtree's leaf descriptor contains different information depending on whether or not the leaf corresponds to a single pixel in the map. If the leaf corresponds to a single pixel, then the fourth subfield indicates how many lines pass through that pixel. Non-WHITE nodes of a larger region contain exactly one line segment and the intercepts of the line segment with the leaf's region are stored here. We have 14 bits to encode each of the intercepts of the line with the edges of the block in which it is contained. We use two bits to indicate which of the four edges of the block the line intersects. The remaining 12 bits indicate the distance from a corner of the block to the intercept (the left corner for the north and south edges, the lower corner for the east and west edges). Thus we are able to handle maps containing blocks as large as 4096×4096 pixels.

The insertion and deletion algorithms for our line representation assume that line segments will be treated as indivisible atomic units. This avoids problems arising from roundoff errors caused by the endpoints of a line segment being changed. This is important, as our representation does not explicitly store the endpoints of a line segment, but rather stores a compact form of the digitization of the line segment. A line segment which originally was long might have a slightly different slope when part of it is deleted. If the entire line segment is deleted and the desired remainder is then re-inserted, consistency of the representation can be insured.

Insertion and deletion algorithms for edge quadtrees are analogous to those of region or point quadtrees. Insertion of a second line segment into a region described by a leaf that already contains one line segment causes the leaf to be quartered. The information that was in the original leaf is distributed among the new leaves and the insertion attempt is repeated. Deletion of line segments is simply a matter of deleting all the information that is specific to that line segment. This means that nodes containing the line segment are given the value WHITE and merged with their siblings if possible. Single pixel nodes have the number of lines passing through them decremented (with the value becoming WHITE if only the delected line passed through the node).

5. DATABASE FUNCTIONS

One of the basic functions of a geographic information system is to indicate what class or polygon contains a given point. Finding the quadtree node containing the point is, of course, a primitive function of the kernel. For most purposes, it is sufficient to describe a polygon simply by listing any point contained within it and its class value. Thus one of the basic database functions is to return a polygon descriptor corresponding to a given point. At times, it is necessary to be able to determine if two points which have the same class value are indeed within the same polygon. For this situation, the user can invoke a function which creates a unique polygon descriptor from a point. This function uses a modified version of the polygon-walking function described in Section 3, examining all of the nodes in the polygon and determining which node has the lowest address. The lower left coordinate of this node is used to describe the polygon. This is an expensive algorithm and would only be used when necessary. Given a polygon descriptor, the class can be determined directly.

The database language allows the formation of a map that corresponds to the extraction of a set of regions from another map. This operation is triggered in the query language (see Section 6) by the key word *map*, and will be referred to henceforth as the SUBSET function. The SUBSET function builds a map which only includes the classes and polygons specified by the user. Alternatively, the user can specify which classes and polygons are to be excluded. The SUBSET function has two phases—one for the class list and one for the polygon list. The class list phase simply traverses the input tree, placing in the output tree any nodes whose class value is on the list. The polygon phase performs a polygon-walking function over each polygon in the polygon list, putting nodes from these polygons into the output tree. The algorithm for the complement of a list of classes and polygons is similar.

At the present we have implemented functions that compute region properties, such as area and perimeter. In addition, we can compute a minimum enclosing rectangle for a given subset of the map, as well as extract a square window from the map. A list of all the classes or polygons from a map can be generated. As an example, such a list could be used to compute the area of every polygon on the map.

Point and line maps can also be used in conjunction with some of these functions, although they may have slightly different definitions. For example, there is no notion of class or polygon. Given the coordinate values of a point, functions are provided to indicate if it lies on a data point or line of the input map. The area of a point map is the number of points contained within it. The area of a line map is the length of the lines within it. A special region search function is provided, similar to the window function, which yields a map containing all of the points within a given radius of a given point from the input point map. The window and enclosing rectangle functions may also be applied to point and line maps.

We have also implemented set operations, such as union and intersection. Both union and intersection may be applied to any two maps of the same type (i.e. both area, line or point). In addition, a line or point map may be intersected with an area map, yielding a line or point map containing those points or lines contained within the non-WHITE regions of the area map.

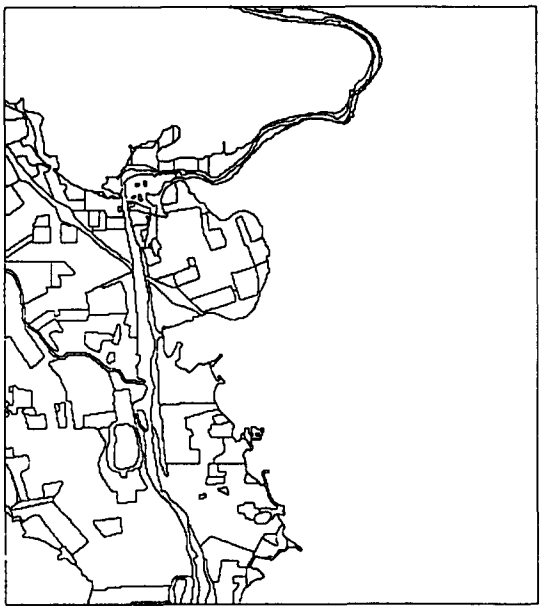

Fig. 11. The intersection of land with the map formed from level1 in top.

6. THE QUERY LANGUAGE

The query language provides an English-like keyword-based interface between the database user and the database system. It allows a non-programming oriented user to access the database with a more natural command language than LISP, thereby enabling him to manipulate maps via the database functions, enter new maps into the system, give names to data items and access the quadtree editor.

The query language is embedded in the University of Maryland version of FRANZ LISP.[18,19] The database system can be viewed operationally as a query language that is interpreted by LISP as LISP functions, which in turn invoke C functions to actually process the maps. In other words, the algorithms of the database are coded in C and LISP merely serves as a convenient front end for translating the query language into calls to C functions.

The query language is keyword-based. It operates by translating a query into LISP function calls, ignoring any words not in its vocabulary. This has the advantage that one can insert noise words and phrases (e.g. articles like 'the' and 'an') to give the command a more natural appearance. Alternatively, one can ignore unnecessary phrases and just type the minimum to cause the appropriate commands to be executed. This added flexibility is bought at the cost of more obscure error messages resulting from the misspelling of a keyword. In order to allow the user to customize his interface with the database, there are commands that allow keywords to be changed.

Table 1 presents a brief syntax of the query language in its present form. The *Please* command is used to learn about the system. The *Use* command changes the display device usage area. The *Measure* command lets the user indicate whether coordinates will have the referred map's lower left corner as origin or use the global coordinate system's origin. The *Enter* command allows the user to inform the system of new data files. The *Display* command enables the display of a map on the display device. It is also used to show the results of any computed function. The *Let*, *Describe* and *Forget* commands manipulate names of entities in the system—e.g. to rename items, describe items or to remove items from the system, respectively. *Let* and *Forget* allow the user to name or forget a data item (e.g. assigning a name to a polygon description) as well as renaming keywords of the query language. *List* returns a list of polygons or classes from a map. *Edit* accesses the quadtree editor. *Move* displays a cursor at a given point on the display device.

One of the key features of the implementation of our query language is the ability to compose functions. Thus, where the *Display* command requires a map, this could be either a map name or an expression which yields a map. For example, if we want to display the intersection of the landuse class map with the region below 100 feet elevation, it could be done with the following command:

Display the intersection of land with the map formed from level 1 in top on the Grinnell

where 'land' is the name of the landuse map, 'top' is the name of the topography map, 'level1' is the elevation class from 0 to 100 feet and 'Grinnell' is the name of our display device. The result of this command is shown in Fig. 11.

Table 1. The syntax of the query language

Commands:

Please {explain} ⟨syntactic_unit⟩ {}
Use {the Grinnell at} ⟨window⟩ {}
Measure {points from the lower left corner of} map {}
Measure {points from the} global {origin}
Enter ⟨file_name⟩ {into database}
Display ⟨map⟩ {on Grinnell}
Display ⟨map⟩ {on Grinnell starting from} ⟨point⟩ {}
Display {the} value {of} ⟨number⟩ {}
Let ⟨name⟩ {} denote {} ⟨object⟩ {}
Let ⟨name⟩ {} rename {} ⟨map⟩ {}
Describe {the type of this} ⟨name⟩ {}
Forget {about the meaning of this} ⟨key_word⟩ {}
List {all the} classes {on} ⟨map⟩ {}
List {all the} polygons {on} ⟨map⟩ {}
Edit {} ⟨map⟩ {with the database editor}
Move {to} ⟨point⟩ {}

Other syntactic units:

⟨number⟩::= {the} area {of} ⟨map⟩
{the} perimeter {of} ⟨map⟩
⟨point⟩::= {where x =} ⟨number⟩ {and y =} ⟨number⟩
{the point at the} cursor
⟨window⟩::= ⟨point⟩ {extending} ⟨number⟩ {by} ⟨number⟩
{the smallest} window {for} ⟨map⟩
⟨map⟩::= {the} intersection {of} ⟨map⟩ {with} ⟨map⟩
{the} union {of} ⟨map⟩ {with} ⟨map⟩
{the} windowing {of} ⟨map⟩ {with} ⟨window⟩
{the} map {formed from} ⟨cplist⟩ {in} ⟨map⟩
⟨class⟩::= {the} class {of} ⟨polygon⟩
{the} class {at} ⟨point⟩ {on} ⟨map⟩
⟨polygon⟩::= {the} polygon {at} ⟨point⟩ {on} ⟨map⟩
{the} unique polygon {at} ⟨point⟩ {on} ⟨map⟩
⟨cplist⟩::= ⟨a list of polygons and classes⟩

Words enclosed in curly braces {} are noise words and may be removed or replaced with any other non-keyword. Words enclosed in angle brackets ⟨⟩ are syntactic units and are replaced by words or phrases matching their definition. In addition to a variable name or integer value which corresponds to the requested syntactic unit in a command, some syntactic units have further definitions, as listed above. For example, where ⟨number⟩ is requested, a number may be typed. Alternatively, one of the two definitions given above for ⟨number⟩ may be used (with the first definition resulting in the area of the map, the second definition resulting in the perimeter of the map).

7. CONCLUDING REMARKS

Our experience in developing a geographic information system based on quadtrees demonstrates that such a system is feasible. The potential advantage of using quadtrees, rather than conventional data structures, lies in the efficiency with which many types of queries can be handled. In its current state, our system can handle a wide range of queries. More capabilities will be added in the future. The system places no restriction on the number of maps which can be placed in the database. For the operations that we have implemented so far, we never use more than two input maps and one output map at the same time. The map size is limited by the address space available, which is a function of the node size. In the current implementation, an individual map may not be larger than 4096 × 4096 pixels. Larger regions can represented by breaking them up into smaller maps.

REFERENCES

1. A. Klinger, Patterns and search statistics, *Optimizing Methods in Statistics*, J. S. Rustagi, ed., pp. 303–337. Academic Press, New York (1971).
2. H. Samet, The quadtree and related hierarchical data structures, *Ass. comput. Mach. Comput. Surv.* **16** (1984).
3. I. Gargantini, An effective way to represent quadtrees, *Communs Ass. comput. Mach.* **25**, 905–910 (1982).
4. D. J. Abel and J. L. Smith, A data structure and algorithm based on a linear key for a rectangle retrieval problem, *Comp. Vision Graphics Image Process.* **24**, 1–13 (1983).
5. D. Comer, The ubiquitous B-tree, *Ass. comput. Mach. comput. Surv.* **11**, 121–137 (1979).
6. A. Rosenfeld, H. Samet, C. Shaffer and R. E. Webber, Application of hierarchical data structures to geographical information systems, Computer Science TR-1197, University of Maryland, College Park, MD (1982).
7. A. Rosenfeld, H. Samet, C. Shaffer and R. E. Webber, Application of hierarchical data structures to geographical information systems phase II, Computer Science TR-1327, University of Maryland, College Park, MD (1983).
8. S. K. Chang, Guest ed., Pictorial information systems, *Computer* **14**(11), 10–67 (1981).

9. R. Barrera and A. Buchmann, Schema definition and query language for a geographical database system, *IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, pp. 250–256 (1981).
10. A. Meier and M. Ilg, Consistent operations on a spatial data structure, *IEEE Computer Society Conference on Pattern Recognition and Image Processing*, pp. 432–440 (1982).
11. D. M. McKeown, Jr, Concept maps, *DARPA Image Understanding Workshop*, pp. 142–153 (1982).
12. P. D. Vaidya, L. G. Shapiro, R. M. Haralick and G. J. Minden, Design and architectural implications of a spatial information system, *IEEE Trans. Comput.* **C-31**, 1025–1031 (1982).
13. R. A. Finkel and J. L. Bentley, Quad trees: a data structure for retrieval on composite keys, *Acta inf.* **4**, 1–9 (1974).
14. J. L. Bentley, Multidimensional binary search trees used for associative searching, *Communs Ass. comput. Mach.* **18**, 509–517 (1975).
15. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Prentice-Hall, Inc., New Jersey (1978).
16. J. A. Orenstein, Multidimensional tries used for associative searching, *Inf. Process. Lett.* **14**, 150–157 (1982).
17. M. Shneier, Two hierarchical linear feature representations: edge pyramids and edge quadtrees, *Comp. Graphics Image Process.* **17**, 211–224 (1981).
18. J. K. Foderaro, *The Franz Lisp Manual*. The Regents of the University of California (1980).
19. E. Allen, R. Trigg and R. Wood, Maryland artificial intelligence group Franz Lisp environment, Computer Science TR-1226, University of Maryland, College Park, MD (1982).

About the Author—HANAN SAMET received a B.S. degree in Engineering from the University of California, Los Angeles, and an M.S. degree in Operations Research and M.S. and Ph.D. degrees in Computer Science from Stanford University, Stanford, CA.

In 1975 he joined the University of Maryland as an Assistant Professor of Computer Science. In 1980 he became an Associate Professor. His research interests are data structures, image processing, programming languages, artificial intelligence and data base management systems.

Dr. Samet is a member of the Association for Computing Machinery, SIGPLAN, Phi Beta Kappa and Tau Beta Pi.

About the Author—AZRIEL ROSENFELD received a Ph.D. in Mathematics from Columbia University in 1957. After ten years in the defense electronics industry, in 1964 he joined the University of Maryland, where he is Research Professor of Computer Science and Director of the Center for Automation Research. He is an editor of the journal *Computer Graphics and Image Processing*, an associate editor of several other journals, a past president of the International Association for Pattern Recognition and president of the consulting firm ImTech, Inc. He has published 15 books and over 300 papers, most of them dealing with the computer analysis of pictorial information.

About the Author—CLIFFORD A. SHAFFER received B.S. and M. S. degrees in Computer Science from the University of Maryland at College Park in 1980 and 1982, respectively. At present he is working towards a Ph.D. degree in Computer Science at Maryland. His research interests include data structures and image processing.

About the Author—ROBERT E. WEBBER received B.S., M.S. and Ph.D. degrees in Computer Science from the University of Maryland at College Park in 1978, 1980 and 1983, respectively. At present he is an Assistant Professor of Computer Science at Rutgers University, New Brunswick, New Jersey. His research interests include geometric complexity, digital geometry and analysis of algorithms.