CAR–TR–308
CS–TR–1886

DAAL02–87–K–0019
July 1987

AN IN-CORE HIERARCHICAL DATA STRUCTURE ORGANIZATION FOR A GEOGRAPHIC DATABASE

Clifford A. Shaffer
Hanan Samet

Center for Automation Research
University of Maryland
College Park, MD 20742

ABSTRACT

The advent of special purpose hardware incorporating large core memories requires the reorganization of databases to take maximum advantage of new processing capabilities. This paper describes the data representation for a geographic system which maintains a large static database on disk with the data of current interest stored in core organized by a pyramid data structure for fast access.

Keywords and phrases: quadtrees, pyramids, spatial databases, geographic information systems

The support of the U.S. Army Harry Diamond Laboratories under Grant DAAL02–87–K–0019 is gratefully acknowledged.

1. INTRODUCTION

The quadtree data structure [Same84a] has received much attention lately as to its usefulness as the underlying data structure for map representation in Geographic Information Systems (GIS) [Same84b, Call86, Garg82, Abel83]. For example, the **QUILT** system [Shaf87b] was designed to support a general purpose geographic database on a conventional VAX-class machine. The advent of special purpose hardware incorporating large amounts of core memory demands a reassessment of the design decisions made when implementing database systems on conventional, relatively limited core memory machines. By making use of the larger internal memory to store additional information, we hope to reduce the amount of disk I/O required when fetching parts of the database to perform a specific operation. Note that this situation is different than that encountered on machines with a large virtual memory. Our experience has shown that it is better to explicitly manage the disk I/O required to access portions of maps rather than to allow the operating system to do it [Shaf87b]. It is important to realize, however, that even in the case where available core memory has greatly increased, it is still not unlimited. While the data structures used for a given application may change, effective use of space is still necessary.

Harry Diamond Laboratories (HDL) is in the process of developing a processor to handle geographic information (GP) [Anto85a, Anto85b]. The GP project aims to deliver a processing unit composed of custom hardware and supporting software that manipulates a geographic database, along with target tracking and planning capabilities. Current specifications for the GP project require support for a static geographic database representing an area 64 kilometers square at 8 meter resolution; this database should contain the current area of brigade operations. The GP processing ability is provided by a small number of independent CPU's (17 68000-based processors are currently planned). These processors will share a common core memory, with each processor dedicated to a specific task.

From the standpoint of the database design for this project, the most significant aspect of the hardware configuration is the unusually large core memory. Approximately 40 megabytes of memory are available in the prototype model. While even this large memory cannot contain the entire geographic database, it should be possible to design a database organization maintaining a significant subset of the geographic information in core at all times. The GP hardware also includes a Raster Technology Model 1/80 display device. An important design consideration is the ease with which the user may display selected features from the database.

This paper examines the use of hierarchical data structures for the GP system. In particular, we examine use of the region quadtree, which decomposes an image into homogeneous blocks. Figure 1 illustrates the region quadtree. If the image is all one color, then it is represented by a single block. If not, then the image is decomposed into quadrants, subquadrants, . . . , until each block is homogeneous. The region of Figure 1a is represented by a binary array in Figure 1b. The resulting quadtree block decomposition is shown in Figure 1c, with the tree structure represented in Figure 1d. This tree structure will be referred to as a *pointer-based* quadtree. This is the traditional method of representing a quadtree, and is most useful when storing an image in core.

The *linear quadtree* [Garg82] is a quadtree storage variant that allows for a reduction in storage space by eliminating the GRAY nodes and pointers required by the pointer-based quadtree. More importantly, the linear quadtree converts the traditional pointer-based tree structure into a sorted list. This allows for the use of standard list processing techniques. In particular, organizing the list by means of a B-tree [Come79] allows for efficient access of the quadtree when stored on disk. Thus, when memory considerations force the quadtree to reside on disk, the linear quadtree has proven to be a popular design choice (this approach is used in [Abel83, Call86, Shaf87b]).

The DF (Depth First) -expression [Kawa80] is a compact representation which simply stores the quadtree's node values in preorder traversal order. It is very efficient in that it does not need to store pointers (which are required by the pointer-based quadtree), nor does it need to store address values (which are required by the linear quadtree). The major drawback when using the DF-expression is that it does not allow random access to arbitrary regions of the map. To find a particular node in the tree, it is necessary to search from the beginning of the DF-expression until that node is reached.

Previous researchers have used the pointer-based quadtree when sufficient core memory was available to store the entire tree. The unusually large amount of core memory available to the GP system suggests that a new approach may be desirable which trades some of this additional space for reduced processing time. In particular, the database system design outlined in this report makes use of a variant of the *pyramid* data structure [Tani80] to improve memory access. The pyramid can be viewed as a stack of arrays which stores at the bottom level the entire image of size $2^n \times 2^n$. At the next level, each disjoint 2×2 pixel block is represented by a single cell, with the entire image at this level represented by $2^{n-1} \times 2^{n-1}$ cells. This process continues until the nth level which contains a single cell.

This paper does not use the term 'pyramid' in its classic sense as a multiresolution stack of arrays. Instead, the pyramid can be viewed as reserving the space required by a *complete quadtree*—i.e., one in which all nodes of the quadtree have been expanded down to the pixel level. This requires $(4 \cdot 2^{2n} - 1)/3$ cells or nodes for the entire pyramid. The quadtree to be represented would be stored within this space. Those subtrees of the pyramid which correspond to a leaf in the quadtree may be ignored during processing. An important aspect of this representation is that node pointers are no longer required. Of more importance, such an arrangement allows for direct access to any node of the quadtree. When using a normal pointer-based quadtree, the nodes can be at an arbitrary location in memory, depending on the interaction between the quadtree building procedure and the node memory allocation procedure. There is no required relationship between the location of a node and its child. Thus, the pointer between them must be traversed, which may require up to n pointer traversals when we need to get from the root to a single pixel-sized node at the bottom of the tree representing an image of size $2^n \times 2^n$. When stored in a pyramid, the location of any node of a given size and position can be directly calculated as an offset from the root of the pyramid. Of course, the user may still need to determine the size of the leaf node containing a specified pixel in the actual quadtree; this process may require searching several levels of the pyramid. However, many applications may be speeded by direct access to nodes based on spatial position. Tucker [Tuck84] used the pyramid as a means of direct access to quadtree nodes when stored on disk, and also investigated efficient neighbor finding operations by means of direct addressing of nodes.

The remainder of this report is organized as follows. Section 2 presents our proposal for organization of the GP's geographic database. Section 3 describes algorithms for traversing and displaying the data stored in the pyramid. Section 4 describes an algorithm for computing the field of view from a given point in the database. Section 5 contains our conclusions and future plans.

2. GP DATABASE ORGANIZATION

This section contains our proposal for the organization of the geographic database in the GP workstation. This database will be divided into five logical parts as follows.

- *The Static Geographic Database (SGD)* contains the complete, high resolution source of geographic data for the GP system. This data is maintained on disk; little modification of this data will be made by the user.
- *The Working Geographic Database (WGD)* contains a subset of the SGD which the user(s) have indicated is of current interest. This data is maintained in core, with pointers to appropriate portions of the SGD on disk. Updates occur when a user specifies that a new feature type is to be examined.
- *The Attribute Database (AD)* contains non-geographic data describing geographic entities. Its organization is unspecified at this time.
- *The Target Database (TD)* contains target sightings and predictions. This data is maintained in core, and is continuously updated by sensor reports.
- *The Temporary Geographic Database (TGD)* contains masks and intermediate products produced on-the-fly by queries to the GP system. This data is maintained in core.

The interaction between these five modules is indicated in Figure 2. During this phase of the project, little attention has been paid to the organization of the Attribute Database; the AD is treated in this report as a "black box." As shown in Figure 3, the

SGD maintains "pointers" to the AD. This does not necessarily mean actual pointers to specific memory locations, but rather some unspecified form of access. One possibility for an AD implementation is to use an off-the-shelf commercial database system.

This proposal concentrates on the design for the Static and Working Geographic Databases. In addition, a data structure to represent maps in the Temporary Geographic Database is proposed. No suggestions for the implementation of the Target Database are made at this time since we feel that we do not yet have sufficient understanding of the operations which this database must support. Design of the TD will be a high priority for future work.

2.1. THE STATIC GEOGRAPHIC DATABASE

The Static Geographic Database design assumes that the geographic database represents a 64 kilometer square to 8 meter resolution. The SGD actually contains a collection of maps. Each map represents a single feature type, possibly with feature sub-types, e.g., a forest map (possibly with differing forest sub-types such as pine and oak) or an interstate highway map. The SGD maintains region, point, and linear features; however, an individual map contains only one of these data types. An organizational sketch for a map in the SGD is shown in Figure 3. Since each map represents a 64 kilometer square at 8 meter resolution (i.e., a resolution of 8192×8192 units), the representation must be compact. At the same time, each map represents only one feature type; thus we can expect homogeneity among adjacent pixels. A variable resolution data structure is therefore desirable. We choose to use the DF-expression [Kawa80] implementation of a quadtree. As mentioned in Section 1, the DF-expression does not normally allow random access to individual nodes. As we shall see, this is not a problem in our implementation.

Associated with each map is a Feature Array (FA). The FA for an area map stores for each feature sub-type a link to the Attribute Database representation of that feature sub-type. For point and linear feature maps, the FA stores a list containing a record for every point or line segment stored in the map. The decomposition of the DF-expression for point features is derived from a PR quadtree [Oren82, Same84a]. Leaf nodes of the DF-expression point to the appropriate record in the FA. For linear features, the decomposition of the DF-expression is derived from the PM quadtree [Same85, Nels86]. Thus, a single leaf node of the DF-expression must be able to store more than one line segment. The varying size required by these nodes to indicate multiple features is handled by reserving one bit to indicate continuation. If the continuation bit belonging to a leaf node N is set, then the next "node" is interpreted as a continuation of N, storing an additional line segment which is contained in N.

Leaf nodes in the DF-expression store an index into the corresponding FA. For area maps, node values need only distinguish the feature sub-type; thus area maps may require only 8 bits per node, allowing 256 distinct feature sub-types in a single map. On the other hand, for point and line maps, the FA stores a description of each point or line feature (plus the continuation bit). Thus DF-expressions for point and line feature maps are given 32 bit nodes to allow for storage of many point and line features in a given map.

The SGD also maintains a table referred to as the Feature Description Table (FDT). This table contains a record for each feature type known to the database. A feature record contains such information as the name of the feature as known to the user; the names of the disk files containing the DF-expression and the Feature Array for that feature type; and the display color and display priority (explained in Section 3) for that feature type.

2.2. THE WORKING GEOGRAPHIC DATABASE

The Working Geographic Database is composed primarily of a pyramid and the Working Feature Table (WFT). The pyramid will contain data to 11 levels (i.e., 1024 by 1024 pixels at the lowest level). Thus, the SGD contains data three levels below that of the WGD. Normally, the WGD will store the top 11 levels of the SGD, i.e., it stores a 64 kilometer square at 64 meters/pixel resolution. However, the WGD data structures are sufficiently flexible that the SGD could be represented in the WGD at a grosser level (for increased processing speed at lower resolution). Alternatively, specified subsections of the SGD could be represented at higher resolution. This topic is discussed further in Section 3.

The pyramid's levels are labeled 0 (for the root) through 10, with identical nodes at all levels (except for level 10, which contains additional information as described below). Each pyramid node contains a bitmap indicating which of the selected feature types are actually contained in that node. Associated with the pyramid is the Bitmap Interpretation Table (BIT). This table contains N records where N is the number of bits available in each pyramid node (i.e., the maximum number of feature types that may be represented at one time). Each record of the BIT describes the feature represented by the corresponding bit position in a pyramid node. An organizational sketch of the WGD is shown in Figure 4.

Each node at Level 10 of the pyramid is augmented by further information (this will be referred to as Level 10^*). Level 10^* contains elevation and slope data, as well as a pointer to the corresponding record in the Working Feature Table for that pixel. Elevation data will be maintained in the database at 64 meter horizontal resolution. The elevation data can alternatively be viewed as being a part of the SGD stored at 64 meter resolution; when creating the initial pyramid, the elevation data is automatically included as part of level 10^*.

A number of design options are available for representing elevation data. This discussion will assume that elevation data is maintained to 1 meter vertical resolution. One option is to simply store 16 bits of data with each Level 10^* pixel. 16 bits of data gives a range of 0 to 65535 meters, much more than necessary. For our application, 13 bits is sufficient (yielding a range of 0 to 8191 meters). This leaves 3 bits which can be used to store slope information. For our application (i.e., primarily trafficability considerations), only a few slope values are necessary. The available 3 bits can represent 8 distinct slope categories. Thus, we can store both raw elevation and slope data at level 10^* for a total requirement of 2 megabytes of storage.

An alternative scheme is to store at each cell in the pyramid the minimum and maximum elevations (or minimum and range) for all child pixels of that cell. Using 32 bits at the higher levels and 16 bits at the bottom level, this will require 3 1/3 megabytes of memory for the elevation data alone.

A third alternative would store the average elevation for the entire map in 16 bits at Level 0 (the root). At the lower levels, each node would contain average elevation for that section of the map. This would continue to the bottom level, where the expected difference between adjacent pixels is smaller. Assuming a maximum difference of 256 meters in the average elevation of adjacent Level 10 pixels (i.e., 64 meter squares), the difference between the pixel's elevation and the average elevation value stored with its father can be maintained at Level 10 in only 8 bits. Thus, the total cost of the elevation data for this scheme would be 1 2/3 megabytes of memory space. One advantage of the third scheme is that average elevation data is stored for all levels of the pyramid. This scheme would not impede the efficiency of slope calculating algorithms (assuming that in this case we prefer to calculate slope on the fly rather than store it explicitly). Calculation of absolute elevation for a Level 10 pixel would require simply adding its difference value to the average value stored with its father.

As mentioned above, each pixel at Level 10* will maintain a pointer to the Working Feature Table. The WFT stores information about each feature contained within the area covered by the corresponding pixel. Records in the WFT are of variable length, reflecting the fact that pixels of the pyramid contain varying numbers of features. The first sub-record of a WFT record contains the length of the record. The remaining sub-records are pointers to features in the SGD which are contained in that pixel. The first few bits of each such sub-record indicate the feature type. This corresponds to a record in the Bit Interpretation Table. BIT records contain the start position in the SGD for the DF-expression containing that particular feature; they also indicate whether that feature type is area, line, or point data. The remainder of the WFT sub-record is the offset into that feature's DF-expression in the SGD. One design alternative for the WFT sub-records would be to add a pointer to the attribute data for each feature's record in the WFT. This would allow direct access from the WGD to the AD. Without this additional pointer, it would first be necessary to fetch the feature's record in the FA (which for area feature types is indexed by the DF-expression).

For each area feature type, the pointer from the WFT sub-record will point to the corresponding node in the SGD's DF-expression for that feature type's map. If this node is a leaf, then the entire pixel will be filled by a single feature of that feature type (or no feature of that type if the node value is WHITE). If the SGD node is GRAY, then the feature's shape will be described by the DF-expression's subtree extending from that GRAY node. Thus, random accessing of the DF-expression is supported by maintaining direct pointers to the desired data. For line or point feature types, it is not necessary that the WFT point to the DF-expression. Instead, the WFT can point directly to the Feature Array record for the given feature. However, this option may require a WFT record to contain several pointers to different individual FA records, since more than one point or a line feature may lie within the area represented by a single pyramid pixel.

2.3. CREATING AND UPDATING THE WORKING GEOGRAPHIC DATABASE

When a user specifies that a feature type is to be added (or deleted) to (from) the WGD, the pyramid and WFT must be updated. Once a bit slot within the pyramid has been selected to represent the new feature, updating the pyramid is a simple matter of traversing the DF-expression for that feature. As each node of the DF-expression is processed, the appropriate bit position for the corresponding cells of the pyramid are turned on or off. Section 3.1 describes two coding schemes for representing a feature type in the pyramid.

Updating the WFT is done by processing the old WFT in parallel with processing of the pyramid (note that the WFT records are stored in the same order as that in which the Level 10 pixels of the pyramid are visited when traversing the corresponding DF-expression). As each pixel is processed, a record is created in the new version of the WFT. For example, assume that we are replacing feature type A in the pyramid by feature type B. As the DF-expression is traversed, the first Level 10 pixel visited will be the upper left pixel of the pyramid. The sub-records for the first record of the old WFT are copied to the new table, leaving space at the beginning of the record to store the total length of the record. Any sub-records dealing with feature type A are not copied. Any features of type B occurring in that node of the DF-expression will have a sub-record entered into the new WFT. Finally, the length field at the beginning of the WFT record is set. When the next pixel is processed, a new record is added to the new copy of the WFT. After the entire pyramid (and the new feature's DF-expression in the SGD) is processed, the new WFT replaces the old version, and the old version's storage is released to the free memory pool.

Note that when the pyramid is constructed, the WFT sub-records for point and linear features point directly to the Feature Array, not to the corresponding DF-expression as for area maps. Thus, it would be possible to eliminate the DF-expression for these feature types entirely. The pyramid would then be generated by processing each line segment (or point) stored in the FA, turning on those pixels which contain the line segment (or point). However, this will involve some calculation to determine which pixels actually contain the line segment. In effect, the DF-expression pre-stores the results of these calculations. In addition, updating the WFT will be much harder since the line features will not be located in the same order as the WFT normally stores pixel descriptions (i.e., in quadtree traversal order). Thus, the DF-expressions for point and linear feature maps are maintained to support efficient construction of the pyramid and WFT.

2.4. THE TEMPORARY DATABASE

It is expected that during operation of the GP system, temporary maps will be constructed to support user queries. For example, one expected query type will list a set of polygons such that the user is to be notified whenever a target enters these polygons. Another query might ask to display all portions of the map visible from a given location, or all portions of the map having a certain set of properties.

Such queries would best be answered by first generating a Mask Map. A Mask Map would be a region map stored in core for efficient manipulation. Since many masks may be active, each must be compact. The mask contains homogeneous regions (and will typically be a binary image), making variable resolution desirable. Random access will be necessary (e.g., to determine if a given point, such as a target sighting, lies in the mask), so a DF-expression is not appropriate. The standard pointer-based region quadtree (as shown in Figure 1d) seems adequate for this task. The linear quadtree [Garg82] is not as desirable since it requires slightly more storage and access time for in-core representations than the pointer-based quadtree [Same86]. After a query is completed, the memory for any associated Mask Maps is freed.

3. PYRAMID TRAVERSAL AND DISPLAY

Pyramid traversal algorithms should take advantage of the hierarchical nature of the pyramid data structure so as to minimize the number of pyramid cells visited. The number of cells which must be visited during a traversal depends in part on how the feature type quadtrees are encoded when stored in the pyramid.

When the GP is initialized, a subset of all feature types available to the SGD are placed in the WGD. Some, but not necessarily all, of these feature types will be displayed. During a work session, the set of features stored in the WGD may be changed, swapping a given feature with another feature from the SGD. It is expected that the SGD will be divided into between 50 and 100 features, and that the WGD will contain up to 32 features at one time.

3.1. ONE BIT/NODE QUADTREE CODING SCHEMES

We can view a single bit position i from all cells of the pyramid as making up a separate pyramid (i.e., if there are 32 bit positions, then 32 individual maps are represented). A single bit is therefore available for each node of a quadtree representation which must describe a particular feature type map. Two interpretation schemes are proposed for encoding these node values, each with relative advantages and disadvantages.

Consider the binary-valued quadtree corresponding to the DF-expression representing a specified feature type. This binary quadtree is made up of internal GRAY nodes and leaf nodes whose values are either BLACK (indicating the existence of this feature type within the node) or WHITE (indicating that this feature type is not contained within the node).

Our first coding scheme turns on the bit value for each pyramid cell which contains the specified feature type. If a given node is marked 1, then any sub-portion of the node may contain the feature. In other words, the GRAY and BLACK leaf nodes of the original quadtree are each represented as 1 in the pyramid; WHITE leaf nodes (and their descendant cells in the pyramid) are represented by 0. For this reason, this scheme will be referred to as GW (GRAY/WHITE) coding.

GW coding yields a straightforward bit interpretation; the disadvantage comes when traversing the pyramid during execution of a process such as image display. When encountering a 0 during the traversal, the node's descendants need not be examined since they must also be 0. The display algorithm may output a WHITE block of the node's size and position. If a 1 is encountered, however, then one or more descendants may contain 0-valued pixels. Thus, the algorithm must repeat for each child. Eventually, every pixel containing the feature must be processed, allowing no effective aggregation of BLACK pixels.

The second coding scheme stores a value of 1 for a GRAY node, and a value of 0 for a leaf node; it will thus be referred to as GL (GRAY/leaf) coding. Since it is necessary to determine the actual value (BLACK or WHITE) of a leaf node, this value will be stored in all descendants of the leaf node. Thus, it is only necessary to examine one of these descendants upon first encountering a 0 value (i.e., the leaf node indicator) to determine the actual leaf value. Note that non-traversal algorithms may be simplified if the single-pixel (bottom) level of the pyramid contains at every pixel the actual value for that pixel (i.e., BLACK or WHITE). In both the GW and GL schemes this will be true, since a value of 1 (whether interpreted as GRAY or BLACK) means that the pixel contains the feature. This is achieved by a slight modification to the GL encoding scheme such that all Level 10 cells store a 1 if the feature type exists in that cell, and 0 otherwise (i.e., the equivalent GW code value). In practice, this modifies the value only for those cells which correspond to Level 10 leaf nodes in the feature type's DF-expression as stored in the SGD. Both schemes would store a value of 1 for those pixels corresponding to a Level 10 GRAY node in the DF expression, indicating that the feature type does appear within that pixel.

The advantage of GL coding over GW coding is that, during tree traversal, full quadtree pixel aggregation is supported. Upon encountering a leaf node of either value, processing may stop. The disadvantages of the GL coding scheme are 1) it is slightly more complicated to interpret node values during traversal and 2) if the pyramid is entered at any level other than the root or pixel levels, the node value may distinguish either GRAY/leaf or BLACK/WHITE depending on the values of its ancestors. The most likely time to examine the pyramid at a middle level occurs after locating a leaf node for a given feature type. At this point, it may be desirable to determine the existence of other feature types at that position in the pyramid. To determine a middle level node's value, it is necessary to visit one or more of the node's ancestors. If the node's bit value is 0, then its true value is WHITE when its parent is 0, and the value of its child otherwise. If the bit value is 1, then multiple ancestors may need to be visited by moving up the pyramid until either a 0 value or the root is encountered. If a 0 value is first encountered, then the node's true value is BLACK; otherwise the node's true value is GRAY. In general, a value of 1 means that at least some portion of the node contains the feature; a value of 0 has the value of its child (or WHITE if at the bottom).

Loading a new feature into the pyramid under either coding scheme is straightforward. Using the GW coding scheme, when a GRAY node is encountered, the corresponding node in the pyramid at that level is turned on (recall that when the DF-expression is GRAY, at least one child of that node must contain the feature). If a WHITE node is reached in the DF-expression, then the corresponding node at that level in the pyramid and all of its descendants have their bit turned off. If a node containing a feature is reached, the corresponding pyramid node and its descendants have their bit turned on. Using the GL scheme, if a GRAY node is encountered, the corresponding node in the pyramid at that level is turned on. If a leaf is encountered, the bit is turned off, and all descendants are set to the value of the leaf node (i.e., BLACK or WHITE).

3.2. DATABASE DISPLAY

As an example of a pyramid traversal algorithm, we will discuss display algorithms. This is particularly interesting since the Raster Technologies display device used in the GP allows display of arbitrary rectangles as a primitive operation. The system architecture is organized such that the display device is on the system bus; it must be accessed by system write commands. Minimizing the number of display primitives written will therefore minimize the (relatively slow) I/O processing required.

Given a specified feature type corresponding to a particular bit plane in a pyramid encoded by the GL coding scheme, the display algorithm is very simple. The pyramid's cells are processed in preorder traversal order. If a node with value 1 (i.e., GRAY) is encountered, then the algorithm recursively visits the node's 4 children. If a node of value 0 (i.e., a leaf) is encountered, then a square of size and position matching that of the leaf is output on the display. The value of this square is that of the node's NW child. No other descendants of the node need be visited.

If the GW coding scheme is used, an efficient display algorithm is somewhat more complicated. Basically, the idea is to initialize the entire display to BLACK (requiring only a single display primitive) and then change to WHITE those portions of the screen corresponding to WHITE nodes. Thus, the number of display primitives is minimized;

however, the number of pyramid cells processed is much larger than that required by the GL coding scheme. The algorithm works as follows. Initially the entire screen is BLACK. If the node encountered contains a value of 1, than no block is displayed and all 4 children of the node are recursively visited by the display algorithm. If the node encountered has value 0, then a WHITE block of the appropriate size and position is displayed; however, no descendants of this node need be processed.

Notice that this algorithm for GW processing actually writes fewer display primitives than the number of leaf nodes in the tree. In fact, the algorithm for either coding scheme can be modified to minimize the number of display primitives. This is done by first initializing the entire screen to the value of the first pixel in the pyramid, and then always keeping track of the current "displayed" pixel value. If the "displayed" value is different than the true pixel value, then the largest block for which this pixel is the upper left corner should be displayed. This technique is used in [Shaf86, Shaf87a, Shaf87c] to greatly reduce the work required to build and manipulate quadtrees. The minimum number of display primitives required will thus be the same for both coding schemes; however, the number of pyramid cells visited will be less for the GL coding scheme.

We now address the issue of displaying a pyramid in which several features are stored. This is a difficult problem to solve efficiently since a given pixel may contain several feature types. We wish to select from among these feature types the one with the highest predetermined display priority. In other words, given a vector of n bits, we need to determine which of the turned-on bits has the highest priority. One way to do this would be to consider each possible bit pattern as an index into a lookup table. The value of the lookup table for a particular pattern would be the color of the bit with priority. Unfortunately, for a 32 bit pyramid cell (i.e., where 32 feature types are stored simultaneously), this would require a table of 2^{32} values. Our alternative approach amounts to a search in a list of n items for the greatest (highest priority) value. Note, however, that the list contains n fixed values, each value in a fixed position. The only factor not fixed is whether a given value is in the list, or left out (i.e., whether or not the corresponding bit is turned on).

There are two basic approaches to storing the feature types in the pyramid with respect to selecting the display bit for a pixel. One approach is to assign feature types to bit slots by order of their display priority. This may simplify determining which feature has display priority (it would correspond to the position of the most significant "1" bit in the word). However, it greatly complicates insertion and deletion of feature types in the pyramid. Both insertion and deletion operations would require that each node of the pyramid be re-ordered to reflect the change. The other alternative is to ignore display priority when assigning feature types to bit slots. This greatly simplifies adding and deleting feature types in the pyramid, since a quadtree with a given display priority may go into any available bit slot.

Fortunately, the best algorithm developed so far to compute display priority does not require that the pyramid nodes be ordered with respect to priority. This algorithm will first be described assuming ordered priority, for clarity. We will then show how it can be modified for an arbitrary ordering. The naive approach to determining priority is to execute what amounts to a sequential search on the bit vector. In other words, look at the first bit, and remember its priority if the bit is on. Then look at the next bit, and remember its priority if the bit is on and of higher priority than the first bit. This continues until each bit of the vector has been examined.

A more efficient algorithm performs a binary search to find the greatest value in the vector. We view the pyramid word containing the bit vector as an integer value. At the first step, we compare the value of the vector (as an integer) to an integer with the first $n/2$ bits on, and the last $n/2$ bits off. If the result of a logical AND operation between the pyramid value and the test value is non-0, then the priority bit is in the first half of the word. At the second step, we check $n/4$ bits (whether in the high half or low half of the word depends on the first step) and so on until we find the priority bit in $\log n$ steps.

If the pyramid cell is unordered with respect to display priority, we simply need to generate test values depending on the display priorities of the bits. The $n/2$ bits with the highest priority would be tested first, and so on. A table of size $n-1$ would be maintained containing the test values; it would be modified whenever the bit priorities change.

From the above discussion we see that calculating priorities is expensive, since this operation must be performed for a large number of pyramid cells. We therefore propose the alternate strategy of avoiding the problem whenever possible. This is done by maintaining at all times a quadtree representing the image corresponding to the display state of the pyramid. In other words, we pre-compute the image that would be displayed by the initial state of the GP. This image is stored on disk and retrieved when the GP is initialized. When features are added to or deleted from the set of features to be displayed, this display quadtree is modified to reflect the current display situation.

Adding a feature type for display simply requires a traversal of the display quadtree in parallel with the DF-expression for that feature type. Those nodes containing the feature type are compared against the display quadtree to determine which value has priority. This requires only a single test, with the tree being modified as appropriate. No priority search through the corresponding pyramid cell's bit vector is required. Deleting a feature requires that the display quadtree be traversed, replacing those nodes which had the deleted feature as priority with the priority value of the remaining features. This requires a search of the corresponding pyramid cell to determine the new priority value.

Another difficulty occurs when the user wishes to change the display window. Recall that the GP database represents a 64 kilometer square. The pyramid stores an image of 1024×1024 pixels at a resolution of 64 meters/pixel. The SGD stores the database at 8 meters/pixel resolution. If the user wishes to rescale, or visit a different section of the map at high resolution, then the display image will change. We can take one of two approaches with the image quadtree. First, we can store a 1024×1024 pixel quadtree representing just what is currently on the display screen. The disadvantage is that when the display window shifts, an entirely new display image, and its associated display priorities, must be recalculated from scratch. The other alternative is to store the display quadtree at the full resolution, i.e., a 8192×8192 pixel image. When the display window is shifted, it is easy to display the appropriate section. The disadvantage of this approach is that when features are added or deleted, more work must be done to update the display image since a larger quadtree is required to maintain the higher resolution data. If features are updated rarely, and display windows change frequently, than this is not a significant problem.

This second approach is also amenable to solution within a multiprocessor system. We can assign one processor the job of constantly reading the display quadtree and updating the display (updates need only be performed if the display quadtree has been modified since the beginning of the last display update). A second processor has the task of updating the display quadtree whenever a feature is added or deleted. This processor would first update that section of the display quadtree within the display window (alternatively, the upper nodes of the quadtree if the display is currently at coarse resolution). Thus, we get the best of both worlds by only needing to wait for the local display window to be updated when a feature is added/deleted, but having a complete display quadtree when the window is shifted. The cost of maintaining the larger quadtree will in most cases be hidden from the user since the additional processing time does not interfere with the display process.

Display window shifting should be restricted so as not to break up quadtree nodes. This will make the process more efficient since less work will be required both to display a node, and to determine which nodes are contained in the window. Ideally, the window should correspond to one, or at most a few, subtrees in the display quadtree. This is accomplished by allowing the window to shift by a multiple of some basic step. For our application, a step of 2048 meters (equivalent to 256 pixels at the SGD's 8 meter resolution, and 32 pixels at the pyramid's 64 meter resolution) might be appropriate. In this case, the image window will correspond to a number of 2048 meter wide nodes in the display quadtree. Such a restriction on the position of the display window should present no serious difficulty to the user.

4. FIELD OF VIEW

This section describes a field of view algorithm for elevation data represented by an array. Since the elevation data for the GP project is stored at level 10* in the pyramid (i.e., essentially in an array format), the algorithm is easily applicable. The field of view algorithm, given a position X, returns a binary array where a pixel value is BLACK if the pixel is visible from X, and WHITE otherwise.

Our algorithm works as follows. A ray is cast from X to each pixel along the edge of the image array. For each such ray, we travel from X out to the border pixel, visiting each pixel C crossed during this process. We look at the value of the elevation for C, and compare the angle formed between C, X, and a point with C's position and X's elevation. If this angle is greater than any angle value for all pixels closer to X along the ray, then C is visible from X.

The field of view operation can be made more efficient by replacing the angle calculation function with a simpler function which is monotonically increasing as the corresponding angle value increases. The actual function used in our implementation divides the difference in height between C and X by the sum of the x and y distances from C to X. If C's value under this function is greater than the value of any pixel between C and X, then C is visible from X.

5. CONCLUSIONS AND FUTURE PLANS

Further results await completion of our implementation, and testing on GP hardware. An empirical comparison should be performed for the bitplane coding schemes described in Section 3.2. The Attribute Database and the Target Database also require further contemplation. However, this work cannot be continued until a detailed description of the target database requirements has been obtained from HDL.

Our database organization is expected to support the implementation of the following algorithms, each of which are either discussed in this paper, or are implemented elsewhere [Shaf87b]: 1) compute all pixels within a given distance of a polygon; 2) compute all pixels visible from a specified point; 3) quadtree image display; 4) subset and set operations on mask maps; and 5) rotation/translation of images. Implementations must also be developed for many functions, including: 1) compute all pixels reachable from a specified point in a specified amount of time; 2) compute all pixels which may be traversed by a given vehicle; and 3) a path-planning algorithm.

6. REFERENCES

1. [Abel83] D.J. Abel and J.L. Smith, A data structure and algorithm based on a linear key for a rectangle retrieval problem, *Computer Vision, Graphics, and Image Processing 24*, 1(October 1983), 1-13.

2. [Anto85a] - R. Antony, Quadtree-based map knowledge for military expert systems, DELHD-RT-RD-85-1, January 1985.

3. [Anto85b] - R. Antony, Spatial, temporal and hierarchical reasoning in battlefield data fusion and scene understanding, DELHD-RT-RD-85-2, May 1985.

4. [Call86] - M. Callen, I. James, D.C. Mason, and N. Quarmby, A test-bed for experiments on hierarchical data models in integrated geographic information systems, in *Spatial Data Processing Using Tesseral Methods,* B.M. Diaz and S.B.M. Bell, eds., September 1986, Swindon, Great Britain.

5. [Come79] - D. Comer, The Ubiquitous B-tree, *ACM Computing Surveys 11,* 2(June 1979), 121-137.

6. [Garg82] - I. Gargantini, An effective way to represent quadtrees, *Communications of the ACM 25,* 12(December 1982), 905-910.

7. [Kawa80] - E. Kawaguchi and T. Endo, On a method of binary picture representation and its application to data compression, *IEEE Transactions on Pattern Analysis and Machine Intelligence 2,* 1(January 1980), 27-35.

8. [Nels86] - R.C. Nelson and H. Samet, A consistent hierarchical representation for vector data, *Computer Graphics 20,* 4(August 1986), 197-206 (also *Proceedings of the*

SIGGRAPH'86 Conference, Dallas, August 1986).

9. [Oren82] - J.A. Orenstein, Multidimensional tries used for associative searching, *Information Processing Letters 14*, 4(June 1982), 150-157.

10. [Same84a] - H. Samet, The quadtree and related hierarchical data structures, *ACM Computing Surveys 16*, 2(June 1984), 187-260.

11. [Same84b] - H. Samet, A. Rosenfeld, C.A. Shaffer, and R.E. Webber, A geographic information system using quadtrees, *Pattern Recognition 17*, 6(1984), 647-656.

12. [Same85] - H. Samet and R.E. Webber, Storing a collection of polygons using quadtrees, *ACM Transactions on Graphics 4*, 3(July 1985), 182-222.

13. [Same86] - H. Samet and R.E. Webber, A comparison of the space requirements of multi-dimensional quadtree-based file structures Computer Science TR-1711, University of Maryland, College Park, MD, September 1986.

14. [Shaf86] - C.A. Shaffer, *Application of alternative quadtree representations,* Ph.D. dissertation, TR-1672, Computer Science Department, University of Maryland, College Park, MD, June 1986.

15. [Shaf87a] - C.A. Shaffer and H. Samet, Optimal Quadtree Construction Algorithms, *Computer Vision, Graphics, and Image Processing 37* 3(March 1987), 402-419.

16. [Shaf87b] - C.A. Shaffer, H. Samet, and R.C. Nelson, **QUILT**: a geographic information system based on quadtrees, Tutorial Lecture Notes 18, SIGGRAPH'87, July 1987. Also to appear as University of Maryland TR.

17. [Shaf87c] - C.A. Shaffer and H. Samet, Set functions for unregistered quadtrees, in preparation.

18. [Tani80] - S. Tanimoto and A. Klinger, Eds., *Structured Computer Vision*, Academic Press, New York, 1980.

19. [Tuck84b] - L.W. Tucker, Computer vision using quadtree refinement, Ph.D. dissertation, Department of Electrical Engineering and Computer Science, Polytechnic Institute of New York, Brooklyn, NY, May 1984.

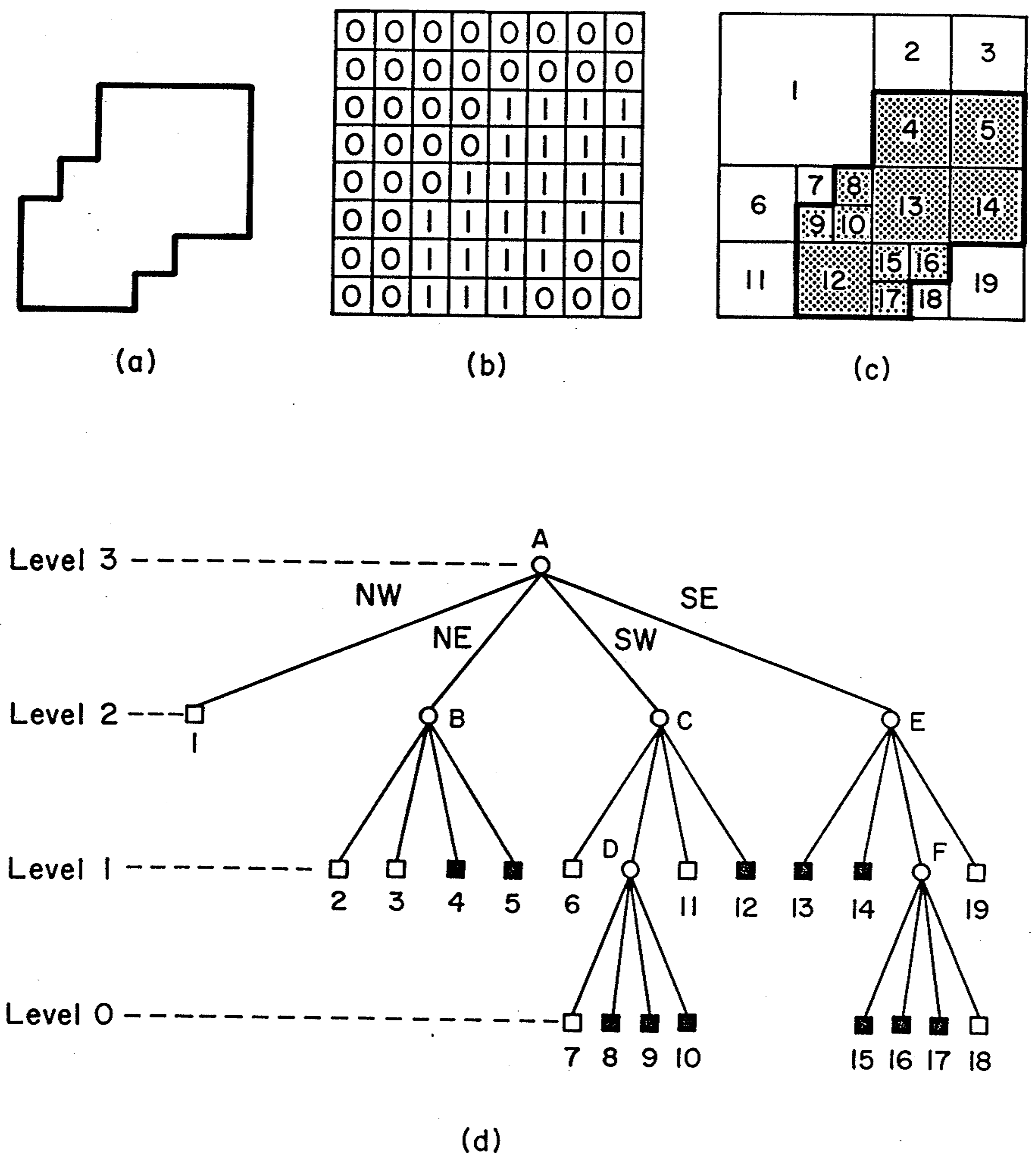

Figure 1. An example (a) region, (b) its binary array, (c) its maximal blocks (blocks in the region are shaded), and (d) the corresponding quadtree.

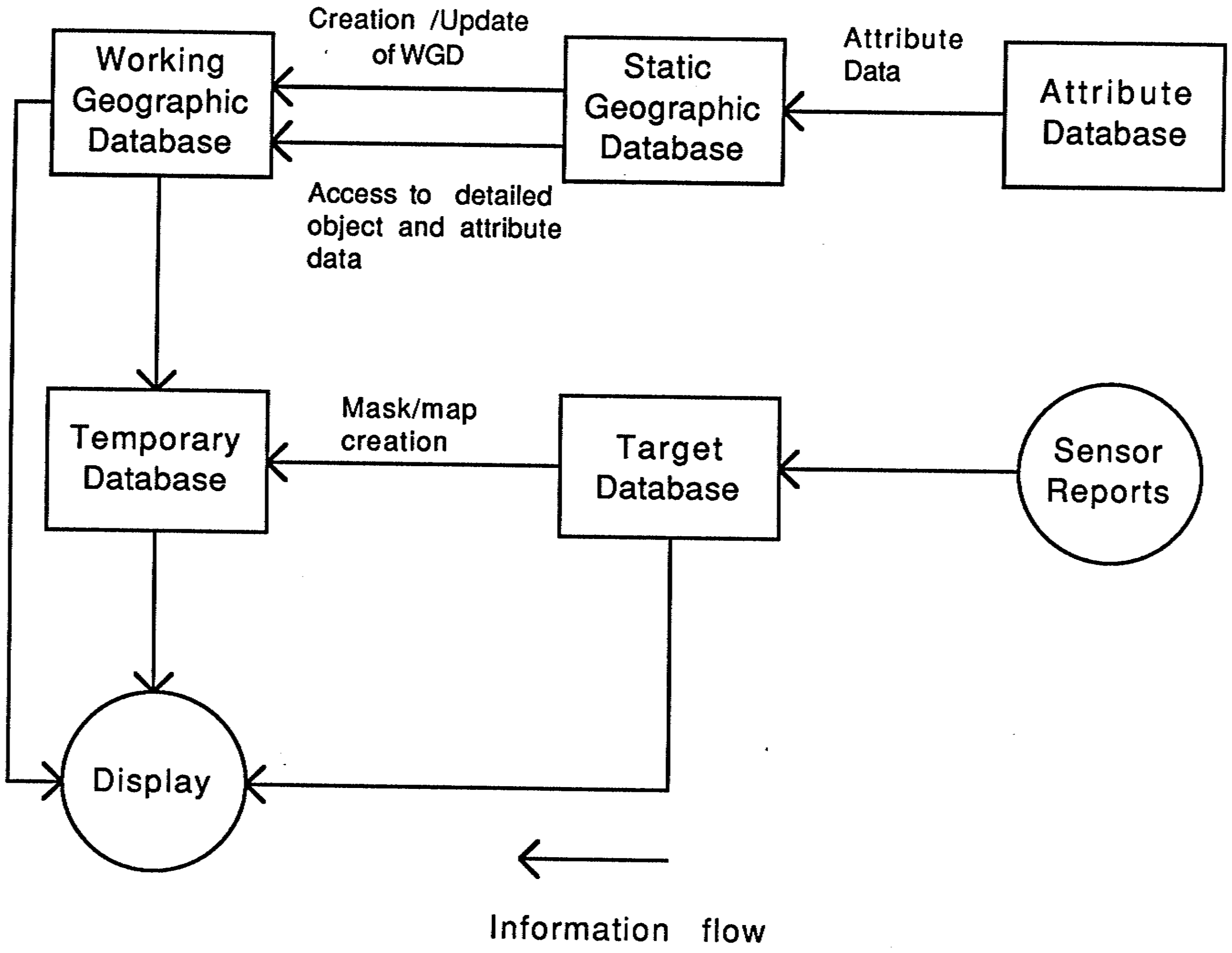

Figure 2. Interaction of CIP database modules

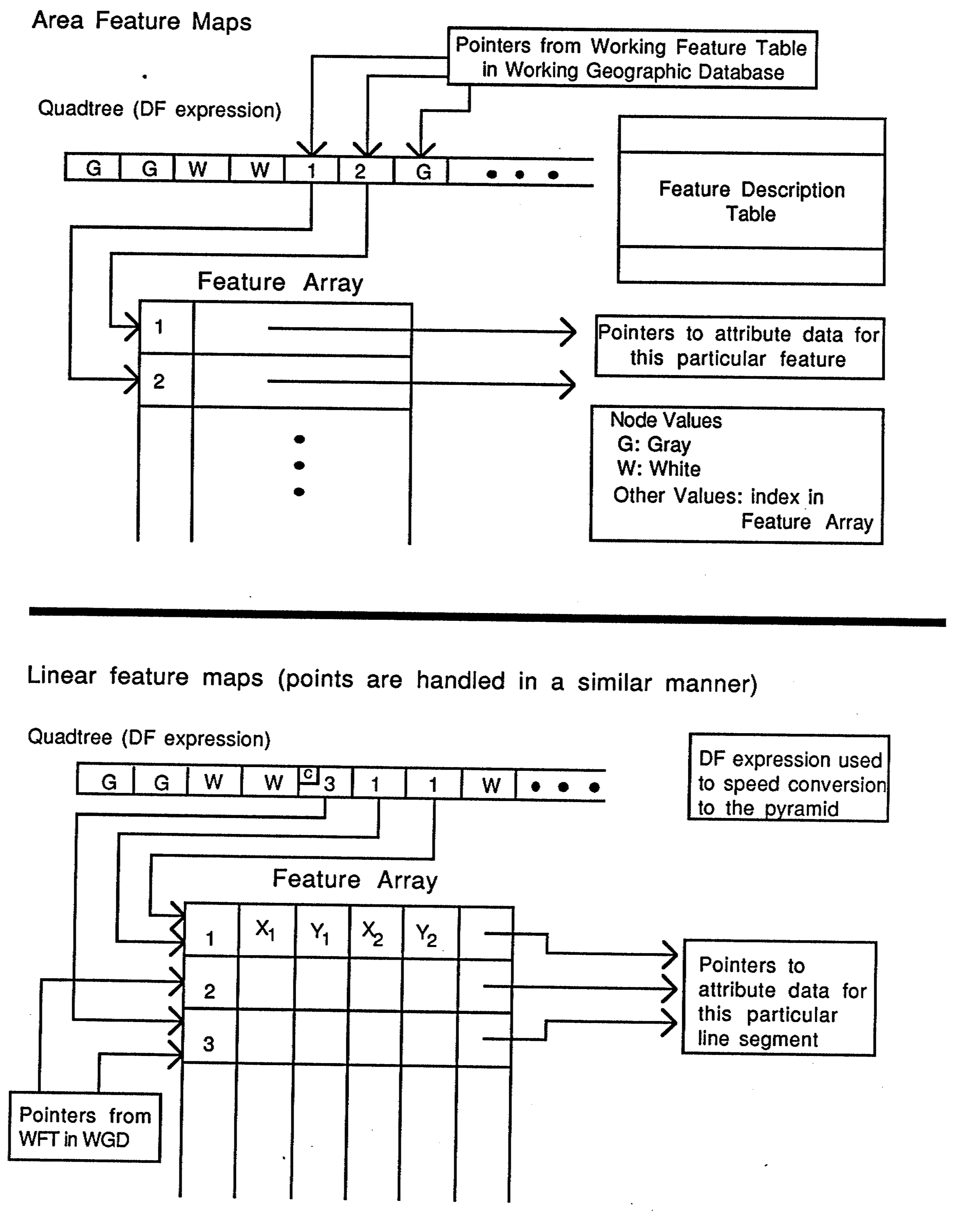
Area Feature Maps
Pointers from Working Feature Table in Working Geographic Database
Quadtree (DF expression)
G
G
W
W
1
2
G
Feature Description Table
Feature Array
1
2
Pointers to attribute data for this particular feature
Node Values
G: Gray
W: White
Other Values: index in Feature Array
Linear feature maps (points are handled in a similar manner)
Quadtree (DF expression)
G
G
W
W
C
3
1
1
W
DF expression used to speed conversion to the pyramid
Feature Array
1
2
3
X1
Y1
X2
Y2
Pointers to attribute data for this particular line segment
Pointers from WFT in WGD

Figure 3. The Static Geographic Database representation for a map.

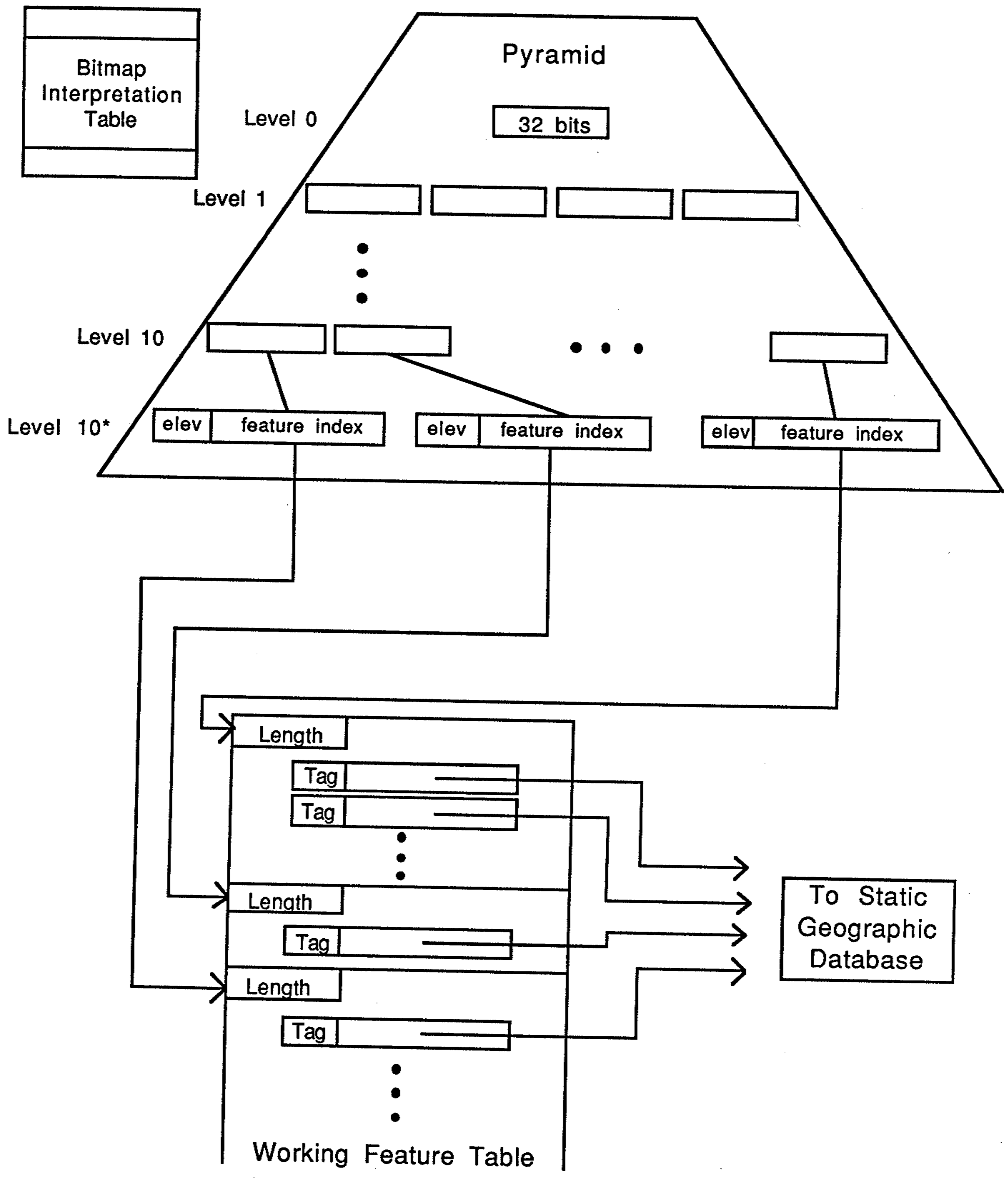

Figure 4. The Working Geographic Database

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CAR-TR-308 CS-TR-1886	5. MONITORING ORGANIZATION REPORT NUMBER(S) N/A
6a. NAME OF PERFORMING ORGANIZATION University of Maryland 6b. OFFICE SYMBOL *(If applicable)* N/A	7a. NAME OF MONITORING ORGANIZATION U.S. Army Harry Diamond Laboratory
6c. ADDRESS *(City, State, and ZIP Code)* Center for Automation Research College Park, MD 20742-3411	7b. ADDRESS *(City, State, and ZIP Code)* 2800 Powder Mill Road Adelphi, MD 20783-1197
8a. NAME OF FUNDING/SPONSORING ORGANIZATION U.S. Army Harry Diamond Laboratory 8b. OFFICE SYMBOL *(If applicable)* ISA/LABCOM	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER DAAL02-87-K-0019
8c. ADDRESS *(City, State, and ZIP Code)* 2800 Powder Mill Road Adelphi, MD 20783-1197	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
An In-Core Hierarchical Data Structure Organization for a Geographic Database

12. PERSONAL AUTHOR(S)
Clifford A. Shaffer and Hanan Samet

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Technical	FROM ______ TO N/A	July 1987	31

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

The advent of special purpose hardware incorporating large core memories requires the reorganization of databases to take maximum advantage of new processing capabilities. This paper describes the data representation for a geographic system which maintains a large static database on disk with the data of current interest stored in core organized by a pyramid data structure for fast access.

Key words and phrases: quadtrees, pyramids, spatial databases, geographic information systems

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE *(Include Area Code)* / 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED