DESIGN AND IMPLEMENTATION OF AN EFFICIENT INDEX STRUCTURE AND A GUI FOR SPATIAL DATABASES USING VISUAL C++

By

SUJATHA SAMADARSINI NEELAM

Bachelor of Technology

S. V. University

Tirupathi, India

1986

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 1995

DESIGN AND IMPLEMENTATION OF AN EFFICIENT

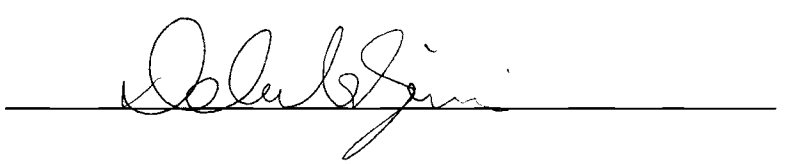
INDEX STRUCTURE AND A GUI FOR

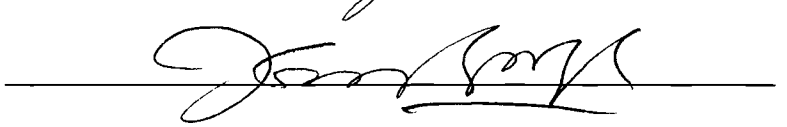
SPATIAL DATABASES USING

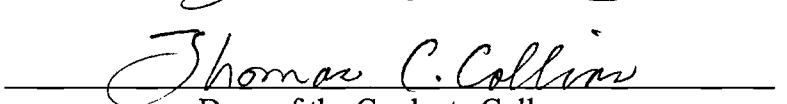
VISUAL C++

Thesis Approved:

Thesis Adviser







Dean of the Graduate College

ACKNOWLEDGMENTS

I wish to express my sincere and special thanks to my major advisor, Dr. Huizhu Lu, for her expert guidance, constructive supervision and friendship. I would also like to thank my committee members Dr. George and Dr. Benjamin for their encouragement, guidance and advice. My sincere thanks to Dr. Mark Gregory of the Department of Agriculture, OSU, for providing the data for this project.

I would also like to take this opportunity to show my appreciation to my brother Mr. VCS Reddy Kummetha for his encouragement, guidance and friendship and heartfelt thanks to our friend Mr. Siva Rama Krishna Kavuturu, for his guidance and help, all through my study at Stillwater, when I am away from my family.

My special appreciation goes to my husband Mr. Chinnappa Reddy Neelam, and my daughter Ms. Pranu Neelam, for all their encouragement, love and understanding. My in depth gratitude goes to my parents Mr. and Mrs. K. Rama Krishna Reddy, for their love, support and encouragement without which I wouldn't have been what Iam today and also for taking care of my daughter, so I can complete my studies successfully. Honorable thanks are due to my parents-in-law, Mr. and Mrs. Neelam Prabhakara Reddy, for their support and appreciation for what I am doing.

TABLE OF CONTENTS

Chapter Page

LIST OF FIGURES AND TABLES

Figures Page

Figures Page

Tables Page

CHAPTER I

INTRODUCTION

In addition to the research being conducted in traditional databases, research is also progressing into non-traditional applications of databases that are often driven by demand. One of such new emerging applications is in Geographic Information Systems (GIS). A Geographic Information System (GIS) is designed to collect, store, retrieve, manipulate and display spatial data or information [FE, 92]. It also represents the new generation of practical applications of hypertext/hypermedia -- which means -- some information may be hidden and a touch of button will reveal all the relevant information of the hidden objects [Fran, 92].

As A.U. Frank says "Space is constructed from objects that fill space" [Fran, 92]. The properties of these objects are defined by a vector of attributes. Collection of these objects is called spatial data. Spatial data consists of points, lines, rectangles, regions, surfaces, and volumes. Since spatial data is very large, data structures that can efficiently handle the huge data have been developed [Good, 92]. There are different data structures like the Quad trees, Octrees, k-d trees, R-trees, k-structure, range trees, Priority search trees, and different types of Quad trees which are used to represent the spatial data. A data structure for a particular application is chosen depending on the operation that is performed on that data structure. A data structure which stores data in core using trees, is different from the one that stores data only in leaf nodes, or for data that resides in non leaf nodes [Same, 92].

Here we consider two-dimensional spatial data related to land. Each point on the land can be referenced with respect to North, South, East and West directions from a

fixed point. Each point also has different attributes such as types of soil, mineral contents, land use (e.g. residential, agricultural, industrial), land value, etc. Data structures such as R trees, R+ trees, R* trees are efficient for storing and managing this type of spatial data. R tree family can hold spatial coordinates as well as their attribute values unlike B-tree which stores an alphabetic key or numeric value in its node, from which they are derived. So a data structure belonging to the R tree family has been chosen as the basic data structure as this proves to be very efficient in storing and accessing huge data when compared to other data structures.

In addition to storing this data a visual representation of data will offer a high level of interaction between the user and the database [Mand, 93]. WYSIWYG (what you see is what you get) representations are good interactive tools since it is easy to locate what is at x, y and to find all objects that are inside a window.

Microsoft Windows provides a good user interface with its powerful graphical tools. A microcomputer is widely used and since the cost of microcomputer is low, a pictorial view of the spatial data based on spatial data structures is a powerful Graphical User Interface (GUI) on MS-DOS machines.

Windows and objects are natural companions. Windows by definition are complex data structures manipulated through a variety of standard operations. In other words windows are objects. Hence Object Oriented Programming is very appropriate for developing Windows [Greg, 91].

In this thesis, the author designs and implements an efficient index structure for spatial data. Furthermore, the author builds a Graphical User Interface (GUI) such as Windows on MS-DOS machines.

In Chapter II the author discusses about two GIS datatypes -- raster data model and vector data model. The author further explores about the merits and demerits of various spatial data structures such as cell methods, Q-trees, k-d trees, k-d-B trees, R trees, R* trees, level linked R* trees which store multi-dimensional data. A general

discussion on Graphical User Interface and Object Oriented Programming infers that OOP is suitable for building a Graphical User Interface.

In Chapter III, the emphasis is mainly on the drawback of the level linked R* tree and a possible solution to improve the level linked R* tree. A spatial index structure for improved access time is also designed.

ChapterIV focuses on analysis of the spatial data, design and implementation of the Windows GUI using Visual C++ for point and range query of the spatial data. This helps the user to query all objects in a window or what is at location x, y.

ChapterV has the summary and the conclusions of the thesis.

CHAPTER II

LITERATURE REVIEW

Most of the information held by government and businesses is geographically referenced. Examples include land use information, mailing addresses, census data, telephone numbers and networks such as cable, water, electric, telephone systems. Infact, geographically referenced information is all around us. So a new field emerged called Geographic Information Systems (GISs). GIS is a widely used tool for collecting, storing, manipulating, displaying or visualization of data that describe space. Organizations use GIS to store data related to space, spatial objects, human activities with reference to space. Spatial data can have information from boundaries and ownership of land parcels to climate data on global scale [FE, 92]. GIS store multimedia data such as images, maps, graphical objects, graphical data and text. In particular, we consider the storage media of spatial data.

Geographic applications present a new set of requirements since objects in GIS are complex. For example, spatial objects such as regions are composed of simpler line objects which are formed by a set of points. With a relational approach these objects can be flattened and can be stored as tables using the traditional data structures, but retrieval is tough and efficiency is reduced. The conventional databases that were developed for commercial systems are not suitable for geographically referenced data. Differences are more prominent especially in areas such as data modeling, query optimization, indexing strategies [LODL, 92].

In this chapter, different GIS data types, spatial data structures, Graphical User Interface and Object Oriented programming are discussed. The following section analyses the GIS data types.

GIS Data Types

Here the GIS data model that needs to be manipulated is briefly analyzed. The classification of geographic data is given in Fig 1. The geographic data can be classified into thematic and spatial data. Spatial data has -- geometric and topological data. Geometric data is description of spatial objects, which can be either in the form of vector or raster format and topological data are spatial relationships between these geometric data. The geometric data represented in vector format is of three types -- points, lines and regions. A line consists of one or more components. A line component can also be represented by connecting set of points. A region is a connected set of lines. Because of the large cardinality spatial relationships are not often stored in topological data. Thematic data is alphanumeric data along with geographic entities such as text and images. Spatial data can be represented both in raster and vector format.

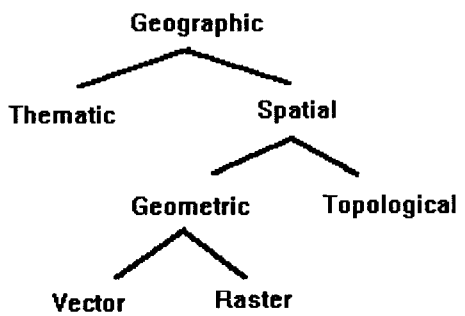


Fig 1. The Hierarchical Classification of Geographical data

Raster data model

This divides space into regular sized and shaped pieces. Attribute values are then assigned to each of these pieces either as averages or as the values at some specific points. The operation on this data model combine data from one raster cell (using the values for different properties) to compute a new data value for the same cell. This is a form of spatial overlay [Fran, 92].

Vector data model

Here space is subdivided into irregularly shaped regions called cells with their boundaries surrounded by lines called arcs or segments that links points called nodes. Most GIS implement operations which map two partitions and determine the intersection area, or to calculate the points of intersection between boundaries. Another implementation uses records for nodes (expressed as coordinate pairs), records for areas

with their attribute values, and records for arcs which contains links to start and end nodes to each arc and links to areas that are to the left and right of arcs (Table 1).

Nodes (node_id, x,y)

Areas (area_id, property_value,...)

Arcs (arc_id, id of start_node, id of end_node, id of left_area, id of right_area)

Table 1. One of the Implementations of Vector Data Model

From the discussion above, although both raster and vector models can store spatial data, vector data model is more suitable for retrieving data having spatial characteristics where as raster data is more image based and hence difficult to instantiate spatial relationships. Hence, the vector formatted data is generally used for manipulation while raster based data is used for visualization. So, generally geometric data in vector format is referred as spatial data.

Spatial concepts consists of ideas, notions and relations between spatial objects which are organized and structured to picture the reality. Spatial concepts differ depending on the task to be performed and the circumstances. A *data model* is a set of objects with operations applied on them. A data model is similar to Abstract Data Types (ADT) defined in Computer Science. A *spatial data model* is a set of conceptual tools used to structure spatial data, including the description of data and appropriate operations. A data model is implemented by selecting a *data structure,* which provides the operations defined for data model, and mapping them onto the code specific for data structure. *Spatial data structures* are low-level descriptors of storage structures and the pertinent operations, with details of achieving the desired results [Fran, 92]. They are also fixed in terms of performance and storage utilization. Many data structures have been proposed

to handle spatial data in GIS [Same, 89b]. The operations commonly performed on these data structures are point query, range query [Knut, 73b]. A *query* is a request for all data that satisfy a predicate or have specific values or ranges of values for specified keys.

Knuth [Knut, 73b] lists queries as follows:

1. A *point query*, which determines if a given data point is in the database and, if so, yields the address corresponding to the record in which it is stored.
2. A *range query* (i.e., region search), which asks for a set of data points whose specified keys have specific values or values within given ranges (includes the partially specified queries, also known as partial match and partial range, where unspecified keys take on the range of the key as their domain).

Spatial Data Structures

The traditional database indexing structures like hash tables, B-trees, ISAM indexes are not suitable for multi-dimensional spatial search. Hash tables are structures based on exact matching of values. Range or nearest neighbor queries cannot be performed since hash tables are designed to destroy order and neighborhood. Since B-trees hold a single key in its node, partial or range queries are impossible as queries involve multiple attributes. Examples of data structures that handle multi-dimensional data are cell methods, Quad trees, k-d trees, k-d-B trees, R trees, R* trees etc. [Same, 90a].

Cell (or fixed grid) methods

Cell (or fixed-grid) methods [Knut, 73b, Bent, 79b] divides space into equal sized cells e.g., squares or cubes for two-dimensional and three-dimensional data respectively, having width equal to the search radius of a range query. The data structure is a directory of k-dimensional array with an entry per cell. A linked list can be implemented to

represent the points in it. A disadvantage of this method is that the structure is not dynamic and cell boundaries have to be decided in advance.

Q-trees

Q-trees [Klin, 76] are used for point data, areas, curves, surfaces and volumes. These data structures are based on the principle of recursive decomposition of space. The resolution of decomposition may be fixed before hand or can depend on the properties of input data. Q-tree is represented in a two-dimensional binary array. The bounded image array is successively subdivided into equal sized quadrants or sub quadrants until each block consist of a 1 or 0; that is, each block is entirely contained in the region or entirely disjoint from it. For large raster images, a second approach is applied. Elements of image are processed one row at a time and the tree is built by adding pixel-sized nodes one by one in a order by which they are in the file. This process is very time consuming due to the many merging and node insertion operations that take place. Besides these trees donot take into consideration of paging of secondary memory. Also, each leaf node occupies large amount of space and node size gets rather large for a k-dimensional tree since $k+2^k+1$ words are needed for each node

PMR quadtree

The PMR quadtree [NS, 86] adaptively maps line segments into buckets of varying size. In two dimensional space, there is a one-to-one correspondence between buckets and blocks. "Spatial occupancy methods decompose space from which data is drawn into regions called buckets" [HS, 92]. A block can have a variable number of line segments. The tree is constructed by inserting a line segment one each into the empty

tree consisting of only one block. Each line segment is inserted into the blocks that it intersects or occupies. During insertion, each block is checked if the insertion exceeds the splitting threshold. If it exceeds, the block is split into four equal blocks.

Deletion from a PMR quadtree is done by removing the line segment from all the blocks it intersects and resides. During deletion, the blocks and siblings are checked to see if the deletion causes the blocks to have the number of line segments below the minimum splitting threshold, then the blocks are recursively merged including its siblings. But, one of the problems is that as the splitting threshold increases, the storage requirements decrease, but the time complexity will increase. The k-d tree is an improvement over PMR quadtree and overcomes the deficiencies.

K-d tree

In the term k-d tree, [Bent, 75b] k denotes the dimensionality of the space being represented. Basically, it is a binary search tree and at each depth a different attribute (or key) value is tested to determine the direction of branching. In two-dimensions (i.e., a 2-d tree), we compare X coordinates at the root and at even depths (assuming root is at depth 0) and Y coordinates at odd depths. Each data point is represented as a node containing six fields. The first two fields are pointers to the left and right children of the node. If P is a pointer and D is the direction, then the field is referred as child (P, D) or LCHILD(P) and RCHILD(P). XCOORD and YCOORD contains X and Y coordinates of data point. NAME field contains information about name (e.g., soil name). The DISC field has name of coordinate that the node discriminates. By convention if node A is an x-discriminator, then all nodes having a x-coordinate value less than A are to its left and value higher than A are to its right, hence DISC field can be avoided. However, a k-d tree node occupies more space and are not inherently parallel data structures since a key

comparison can be performed parallel for the k key values. Like Quad trees, these trees also donot consider paging of secondary memory.

K-d-B tree

The k-d-B tree [Robi, 81] is basically a k-d tree which dictates the contents of B-tree node. The problem with this technique is that B-tree performance guarantees are no longer valid. For example, pages are not guaranteed to be 50% full without complicated insertion and deletion algorithms. The nodes of k-d-B tree correspond to disjoint regions. These trees are useful for point data. The R tree of Guttman [Gutt 84] is another adaptation of B tree that doesnot require the regions covered by nodes to be disjoint.

R tree

The R tree [Gutt, 84] is based on a B+ tree structure and store multi dimensional objects (rectangles). A non leaf node contains minimum bounding rectangles and pointers to its child nodes. A minimum bounding rectangle of a node is one that has minimum area and includes all rectangles that are the entries in its child node. Leaf nodes contains rectangles and data objects related to that rectangle. Fig. 2a represents the rectangles in R tree, and Fig. 2b., represents the corresponding R tree.

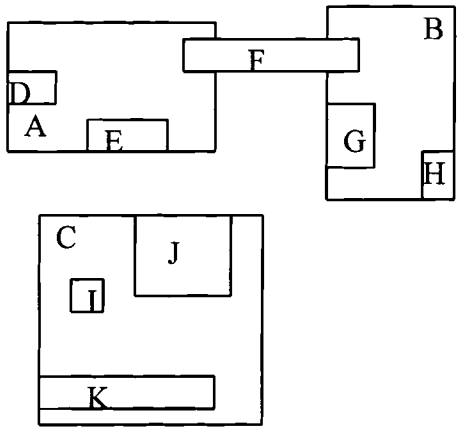


Fig 2a. Rectangles organised in R tree

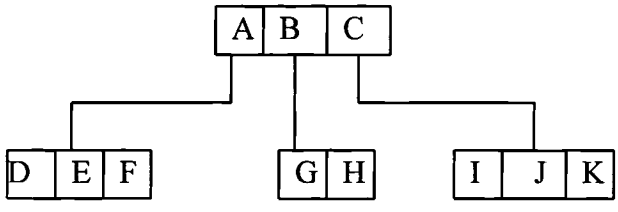


Fig 2b. Corresponding R tree

If 'M' is the maximum number of rectangles that fit in a node and 'm' is the minimum number of rectangles ($2 <= m <= M$) then R tree has the following properties

1. The root has atleast two children unless it is leaf.
2. Every non-leaf node has between m and M children unless it is root.
3. Every leaf node has between m and M index records unless it is root.
4. All leaves appear on same level.

The R tree is a dynamic structure which handles data objects in several dimensions. The bounding boxes are formed from arbitrary set of rectangles in a way that arbitrary retrieval operations with query rectangles of arbitrary size are handled efficiently. The known parameters of retrieval performance affect each other in a very

complex manner which makes optimization impossible without influencing the other. This may cause deterioration in overall performance. Since the data rectangles may have different size and shape and directory rectangles may grow or shrink dynamically, the success of methods that optimize one parameter is very uncertain.

R+ tree

R+ trees [SHS, 86] can be considered as the extensions of K-D-B trees to cover not only points but rectangles also. R+ tree is a variation of R tree and avoids overlap of rectangles, but occupies more space, thus increasing the height of the tree. The main difference between the two data structures is that, rectangles are split into smaller rectangles in order to avoid overlap among minimal bounding rectangles. For example, if there is a rectangle covering a spatial object at the leaf level overlaps with another rectangle, the rectangle is decomposed into non-overlapping sub-rectangles. All the pointers from the sub-rectangles point to the same object. The same method is applied to the non-leaf nodes also, thus overlap is forced to zero. Detailed description is given in the figures 3a, and 3b.

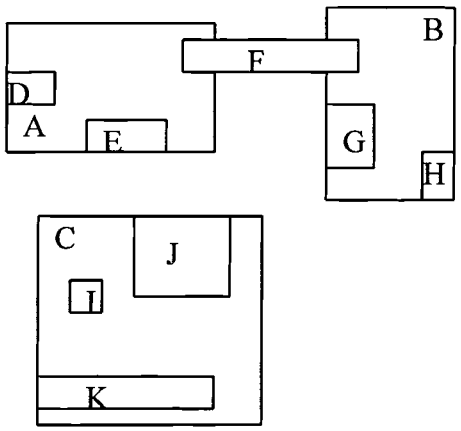


Fig 3a. Rectangles organised in R+ tree

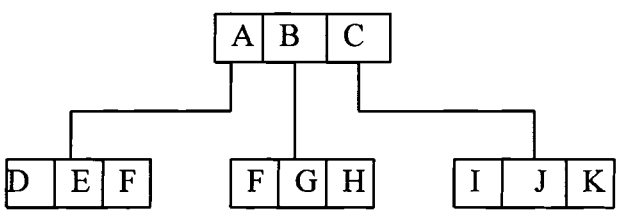


Fig 3b. Corresponding R+ tree

Fig 3a represents all the rectangles that constitute a R+ tree, and Fig 3b is the R+ tree itself. To access rectangle F one can take the path from A or from B from Fig 3b. This is not the case in R tree, as one needs to traverse the whole tree in order to access a particular rectangle. Although accessing a particular rectangle is relatively fast, the height of the tree is increased in the case of R+ tree. For non-pathological split cases, if the number of sub-rectangles at a node increases, this causes a split to that node, thus increasing the height of the tree.

Christos Faloutsos, Timos Sellis and Nick Roussopoulos, analyzed R trees and R+ trees in their paper. On comparing both the data structures, they concluded that "R trees suffer in the case of few, long segments, which force a lot of 'forking' during the search. The R+ trees handle these cases easily, because they split these long segments into smaller ones" [CTN, 87].

R* tree

A heuristic approach is applied taking various parameters into consideration. The parameters that are taken into consideration by the *R* tree* are [NHRB, 90]:

1. Minimize the area covered by a directory rectangle. The dead space in the directory rectangle not covered by any of its child rectangles is minimized.
2. Minimize the overlap between directory rectangles that decreases the number of paths to be traversed.
3. Minimize the margin of the directory rectangle. Margin is the sum of the lengths of the edges of a rectangle. For an fixed area square has the maximum margin. Thus minimizing the margin yields more quadratically shaped directory rectangles. This results in more packed directory rectangles. Queries with large query rectangles profit this.
4. Optimize storage utilization. The higher the storage utilization, the lower the tree height and the better the querying.

Unlike R tree that take only area parameter into consideration R* tree takes area, margin and overlap. The overlap of an entry is defined as [NHRB 90]:

Let E1,...,Ep be the entries in the current node. Then

Overlap(Ek) = area(Ek.Rectangle $\cap$ Ei.Rectangle), $1 \leq k \leq p$.

To insert a new rectangle *choose subtree* function is invoked to find an appropriate node.

Algorithm Choose Subtree

Step1: Set N to be the root

Step2: If N is a leaf, Return N

Else If the child pointers in N point to leaves [determine the minimum overlap cost], choose the entry in N whose rectangle needs least overlap enlargement to include the new data rectangle. Resolve ties by choosing the entry whose rectangle needs least area enlargement then the rectangle with smallest area.

End

Step3: Set N to be the child node pointed to by the child pointer of the chosen entry and repeat from Step2.

From the above algorithm, the subtree is chosen and the node is selected in which the new entry is to be inserted. If the node has less than M entries, the new entry is inserted in that node. If it has M entries algorithm *split* is invoked.

Algorithm split in turn calls two more algorithms:

1. Choose split axis

This chooses the axes along which the split has to be performed by computing various goodness values.

2. Choose split index

This selects the distribution of entries into two groups.

Along each axes the entries are first sorted by the lower value, then by the upper value of the rectangles. For each sort M - 2m + 2 distributions of the M + 1 entries into two groups are determined where the k-th distribution [k = 1,...,(M-2m+2)] has first [m-1=k] entries in the first group and the remaining in the second group. For each distribution the following goodness values are determined:

(1) area-value: area[bb(first group)] + area[bb(second group)]

(2) margin-value: margin[bb(first group)] + margin[bb(second group)]

(3) overlap-value: area[bb(first group)] $\cap$ area[bb(second group)]
where bb represents the bounding rectangle.

Algorithm Split

Step1 Invoke choose split axis to determine the axis, perpendicular to which the split is performed.

Step2 Invoke choose split index to determine the best distribution into two groups along that axis.

Step3 Distribute the entries into two groups.

Algorithm Choose Split Axis

Step1 For each axis sort the entries by the lower and by the upper values of their rectangles and determine all distributions as described above. Compute S, the sum of all margin values of the different distributions.
End

Step2 Choose the axis with the minimum S as split axis.

Algorithm Choose Split Index

Step1 Along the chosen split axis choose the distribution with the minimum overlap value. Resolve ties by choosing the distribution with minimum area value.

Once the split is performed the tree structure has to be updated along the insertion path. All the covering rectangles have to be adjusted such that they are the minimum bounding boxes enclosing their children. Though the insertion is costly, it provides an order in the structure which contributes for fast accessing.

Experiments conducted found that R* tree out performs the R tree variants in all experiments [NHRB, 90]. The conclusions of the experiments are:

1. The R* tree is the most robust method, underlined by the fact that for every query less accesses are required than by any other variants.
2. The gain in efficiency of the R* tree for small query rectangles is higher than for large rectangles because storage utilization becomes more important for large query rectangles. This emphasizes the goodness of the order preservation of the R* tree.
3. The maximum performance gain of the R* tree taken over all query and data files is in comparison to the linear R tree taken over all query and data files is in comparison to the linear R tree about 400% and quadratic R tree is 180%.
4. R* tree has the best storage utilization.

Comparative study of Data Structures

A comparative study was performed by Hanan Samet and Erik G. Hoel on the performance of three spatial indexing structures -- the R* tree, the R+ tree and the PMR quadtree. The data from the TIGER/Line files used by the Bureau of the Census to deal with the road networks in the US, is used as test data. The authors state that their goal is not to find the best among them which is generally impossible, but they are comparable and that an indication can be given as to why and when their performance differs.

Researchers deal with spatial data by mapping the spatial object into a point termed as representative point. Using a representative point, each line can be represented by its end points, which means each line segment is represented by a tuple of four items i.e., a pair of x coordinates and a pair of y coordinates. But this mapping is good for storing data, not ideal for spatial operations involving search. Hence the authors believe that using data structures that are based on spatial occupancy is the best way to overcome

the problems. Spatial occupancy methods decompose the space from which the data is drawn into regions called buckets. Grid files, which use the bucketing methods, deal with the transformed data i.e., representative points. But the authors were interested in bucketing methods that are applied to the space from which the data is drawn. And hence the authors conducted a study on the performance of three popular spatial indexing techniques -- the R* tree, the R+ tree, the PMR quadtree.

Using the R* tree, the R+ tree and the PMR quadtree, 1000 tests were performed for each query type and map. Tests were performed on 6 maps of counties in Maryland where each map contained approximately 50,000 line segments. The disk accesses for all three structures were comparable and the authors found that the PMR quadtree required the least number of disk accesses. And the authors also feel that the number of disk accesses donot give the whole building process of the trees. This is because there is considerable amount of activity such as optimization of the splits in R* tree which is not present in R+ tree or in PMR quadtree. Also, because of the disjoint decomposition of space induced by R+ tree, it is usually better than R* tree. On a query by query comparison, R+ tree was superior, but for the repeated application of the point queries, the R* tree was slightly better. The authors feel that this is not surprising since the R* tree occupies less space than R+ tree.

The authors concludes that although the performance of R* tree is not as good as the R+ tree, R* tree is more compact that R+ tree. They also state " Not surprisingly, our studies did not result in claims of overwhelming superiority for any of the data structures. Qualitatively speaking, they are similar. In terms of choosing a representation for a

specific application, the choice can only be made once the repertoire of operations that is to be executed is known."[HS, 1992]

Level linked R* tree

Some work has been done on R* tree by making it leaf level linked. This level linked R* tree has all the properties of R* tree data structure. A special property is that the leaf nodes are all linked together sequentially. Point or range query is done in the same way as in R* tree, while sequential access to the data is done at the level linked leaf nodes. In R* tree new nodes are created in two cases:

1. When a node is already filled with maximum elements, a new element insertion will lead to a 'split'. When a split occurs a new node is created, linked to its parent and becomes the sibling to the old node. This newly created node can be at the leaf level or at the non-leaf level.
2. When the root node splits, two nodes are created and one of them becomes the root, the other becomes a sibling to the old root.

In case of leaf level split in level linked R* tree, an algorithm was developed by Reddy[93]. Whenever a leaf node splits, the new node's sibling pointer points to the sibling of the old node, and the old node's sibling pointer points to the new node. This modified split algorithm decreases the time complexity though it takes an additional memory requirement of the size of pointer per node [Reddy, 93].

Graphical User Interface

To begin, it is important to examine the role of the interface as a medium of human communication. Computer software differentiates from other media forms such as television, radio etc., by being more interactive and favors participation over

observation. Additionally, software favors order from randomness, reaching the goal of managing complexity.

But, in the computer culture, software also favors cryptic symbolism in order to achieve conservation of expression and C programming language is the example. The burden of maintaining the models and abstractions is forced on the users while, the GUI software opts for visual expression to conservation of expression on the same familiar grounds. The software culture embodies structure and organization rather than randomness, while GUI places more emphasis on organization.

The familiar style of pull-down menus placed in a horizontal manner on the top of the screen is part of this culture. Recognizable, consistent graphical elements such as scroll bars, list boxes and buttons are expressions of this culture. But then, having these elements in the software does not make it a GUI. A software is called a GUI depending on how these elements are significantly organized and visually represented.

In the environment where people have to learn and use software products, the GUI culture saves the effort and the frustration by having recognizable visual elements and sharing an organizational similarity, irrespective of whether the applications are developed in Windows, OS/2 Presentation Manger, Motif, Open Look and the Macintosh [OH, 91].

Application packages provide ways of manipulating things (called objects) and these things are presented in the client area of main window of any GUI. For example, beneath the main menu bar of any word processor is the text. Text ordered as words, sentences, paragraphs can be manipulated by the user and are called 'objects'. All menu choices specifically represent an operation that can be performed on the text that is in the main window. Since the menu system provides a convenient way to explore all the functions it provides, it breaks the initial barrier even for a beginner. The first and

foremost principle of designing menus is to make the user understand that the role of menu is of a complexity browser. A well designed and fully functional menu will encourage the user to examine the available options to gain an insight into the capabilities of the software.

GUI, along with being a good medium of communication, also has another dimension of presentation, which the character-based text does not have -- graphics. Good visual design helps in organizing and prioritizing information, guiding users from general information to specific windows or buttons, with the minimum effort. Good design reduces the randomness of visual information. The more efficient the data gathering process, the more control users have on the data and knowledge of how best to use it. To achieve this result, the visual design of all GUI application must include hierarchies as well as alignment, shape, color and fonts.

Consistency has to be maintained in order to properly receive the visual hierarchy. All information of equal value has to be maintained in the same hierarchy. For example, greyed text implies that the current information is not available to manipulate. Shape has a very high effect and when used along with proper alignment and size, can create an effective visual ordering system. Icons and tools in a tool bar should have related groups of tools shaped in the same manner and placed together. Color adds supplementary information and is a good tool to differentiate between similar objects, although it is not a universal ordering system. Thus the GUI culture makes demands for consistency, efficiency, and the presentation of information in its application software.

With the above general discussion on GUI, to graphically represent the spatial data, we need a sound graphical interface and user friendly environment. User interface forms an essential component when dealing with graphical data, because of the nature of manipulated data [Vois, 91]. Generally, the design of a friendly user interface is a way to

offer the end user a high level of interaction with the database. The graphical language MS Windows can provide a WYSIWYG (what you see is what you get) user interface since it does not restrict the window to any pre-defined size or structure. The user can have the choice of designing the size of window containing a map, as well as the size of map displayed inside the window. A good interface will assume little about the manner in which users will attempt to employ the capabilities of software, especially in regard to sequencing of operations and MS Windows belong to this category [NK, 92]. Considering the fact that MS-DOS machines are extensively used, MS Windows provides a powerful user interface. They donot enforce a particular policy i.e., doesnot provide scroll bars, button boxes, menu etc., by default. Applications can create their own decorations like title bar, scroll bar, menus etc., depending on the need of the user. Windows also have resources like fonts, colors, size of windows etc. There are different styles used in Standard Windows environment such as

1) Overlapped Windows
2) Owned Windows
3) Pop up Windows
4) Child Windows.

Object Oriented Programming

Object Oriented Programming (OOP) is probably the best way to build and maintain a GUI application. Object oriented tools provide powerful libraries of reusable code and the glue that makes them all work together. The code can be used as is or customized to handle the constructs. The power of the code is that it describes not only the object structure but also its behavior and its interface to other objects [OH, 91]. Object tools come with WYSIWYG editors. A class encapsulates (or brings together) data structures and the methods that manipulate them. So, once an object of a particular

class is created, it inherits all its methods and attributes. Hence a composite subclass can be created from multiple parent classes and inherit their combined behavior.

Object Oriented Programming (OOP) is not only playing a major role in developing Windows programming, but is also causing a revolution in the way algorithms and data structures are constructed. OOP is finding a way into applications that involve spatial data [KA, 90]. OOP is the new programming methodology and demands a radical change in designing programs. Instead of maintaining a set of different procedures, where some procedures act on some data structures and some act on others, OOP bundles a set of data structures with the procedures that act upon them. A procedure designed for a structure cannot be applied to a different procedure for which it is not designed for, even though the structures are very similar that their respective procedures have identical parameter call and lists. Also, if a programmer has carefully designed procedures for complicated data structure, it is not possible for anyone to ignore the procedures and manipulate the data structure directly [Vert, 92].

If a new structure is designed along the old structure, the programmer is not demanded to impose the old structure features on to the new, the implementation can be entirely different. The new structure can be defined in terms of the old, with only 'newness' added in. If, the new structure has routines that has the same name as the old routine has, the structure should be able to distinguish them, based on the context of how they are invoked.

The major contributors of OOP are *encapsulation, inheritance,* and *polymorphism* Encapsulation provides information hiding, inheritance provides code reusability and extensibility, and polymorphism provides overloading of similar operation onto procedure names.

Encapsulation provides some data type abstraction. A group of related procedures is bundled into 'module'. Only those, who have explicit access to affect the contents of the module or to obtain information, can do so. OOP allows procedures to be

bundled with data structures and are called 'methods'. These methods work only with the data structures that are bundled with, and hence called 'objects'. Encapsulation provides security in writing programs, and the data structures are not accidentally deleted.

Inheritance allows code reusability. For example, an object was already defined, and later on, if it was found that the object works fine for the present problem, but needs some modification for another. The programmer would then define another object, with all the necessary properties, but with the old object forming the base class, which means, that the new object will 'inherit' all the properties of the old object and as well as have its own exclusive properties. Many levels of objects can be created, each deriving properties from a level above.

Polymorphism allows methods in derived objects to have identical names as the base object has, and the execution of the correct procedure is determined from the context of the object at run-time. This is an example of name loading. The example of operator overloading -- "+" operator, which means, either addition of integers, or addition of reals, or concatenation of strings can be done, but which depends on the context.

CHAPTER III

DRAWBACK OF LEVEL LINKED R*TREE AND SOLUTION

The R* tree is based on B tree where the data is stored in the leaf nodes. To sequentially access data, the whole tree has to be traversed. In order to overcome this difficulty, level linked R* tree data structure was designed and implemented. The level linked R* tree was derived from R* tree. The tree structure is shown below.

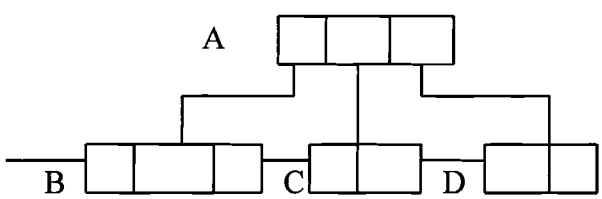


Fig 4. Level linked R* tree

All leaves are linked together as shown in the figure 4. This makes the sequential access to the actual data, faster, than the sequential access in R* tree. But, if an application has a very huge data occupying megabytes of memory, accessing data using this type of level linked data structure could also be time consuming. In this particular application the data is very huge consisting of about 20,000 rectangles. For example, the data could be present in the first few nodes and then in the last few nodes, or the data could be residing only in the first few nodes or the last few nodes or could be spread evenly in all the leaf nodes. With the above data structure leaf linking has to be done by

traversing through all the nodes which donot have relevant information. This could be very time consuming.

A Solution

To overcome the problem encountered in the leaf level linked R* tree data structure, the author has used a leaf level multi-linked R* tree data structure. The exact implementation of the data structure is discussed below.

Since the application is dealing with agricultural data (detailed description and analysis of data is done in next Chapter), each soil is given an attribute. Insertion of an element into the appropriate node is done according to the R* tree insertion algorithm. In addition to that, the element (the attribute value here) is checked against the attribute_array. If this is a new insertion, a link is placed from the appropriate index of attribute_array to this element in the node and the forward link is pointed to NULL. If a link exists already, from this particular index of attribute_array, this link is traversed and the last element that is connected to this link is reached. A link is then placed from this 'last element' to the newly inserted. With this improvement in the data structure, sequential accessing can be done by traversing through the nodes which have only the relevant information. Thus the time consumption is reduced, although it occupies a few bytes of memory for the links and the attribute_array. Considering the amount of time consumed, one can definitely afford a few bytes of memory.

Leaf level multi-linked R* trees:

For applications that need a Graphical User Interface built for spatial data, it is necessary to make this improvement to the basic data structure. Along with the range

query, this improvement of data structure makes the sequential access very efficient. Insertion is done according to the basic R* tree algorithm.

Algorithm Choose Subtree:

S1 Set N to be the root

S2 If N is a leaf, Return N

Else if the child pointers in N point to leaves [determine the minimum overlap cost], choose the entry in N whose rectangle needs least overlap enlargement to include the new data rectangle. Resolve ties by choosing the entry whose rectangle needs least area enlargement than the rectangle with smallest area.

End

S3 Set N to be the child node pointed to by the child pointer of the chosen entry and repeat from S2.

From the above algorithm, the appropriate subtree and the node is chosen in which the new element is inserted. Now another function called *'attribute_link'* is invoked to find the link to this element.

Algorithm Attribute_link:

S1 Pick the index in the attribute array, with the element as the index.

S2 Pick the link from the array index, until NULL is reached. Now place the appropriate link to the newly inserted element.

From the above two algorithms, the element is inserted in its place and the appropriate links are attached, only if there are less than M entries in the node. If it has M entries, then algorithm 'split' is invoked.

Algorithm Split:

S1 Invoke choose split axis to determine the axis, perpendicular to which the split is performed.

S2 Invoke choose split index to determine the best distribution into two groups along that axis.

S3 Distribute the entries into two groups.

S4 If the split node is a leaf node, update the links. Set new node's NEXT pointer to the old node's NEXT pointer, and the old node's NEXT pointer to the new node.

The implementation of the algorithm is:

As soon as the first node (root as well as leaf) is created, the attribute array is checked and the appropriate position of the attribute in the array is picked. A link is then placed to the element in the node from this array.

If there happens to be another element with the same attribute value, a link is set from the old element to this new element. As the tree grows, links from the attribute array are placed to the leaf nodes appropriately and from there the linked list can be traversed. For example, if the soil with attribute value of 36 is to be accessed, the algorithm checks if there exists any link from this position in the attribute_array and traverses the linked list related to attribute value of 36, thus omitting traversing all the elements in all the nodes and saving the time consumed.

Time complexity of leaf level multi-linked R tree:*

In R* trees, to access actual data residing in the leaf nodes, the whole tree has to be traversed. This takes enormous amount of time if the tree is very large. If there are N nodes (assuming there are more than 1 nodes) in the tree, and if it takes one unit of time to access each node in the tree, the time to access the actual data would take more than N time units, since one has to access the leaf nodes and traverse the whole tree.

In leaf level linked trees, where all the leaf nodes are linked together, the time to access the data is greatly reduced, since only the leaf nodes are to be traversed. Now suppose there are 'n' leaf nodes in the tree, and to access the data ,only 'n' units of time is required, assuming that accessing each node takes one unit of time. But, in leaf level multi-linked tree, links are maintained to only those nodes, that maintain information related to on particular attribute -- in this application, soils. So, in order to access the data related to one particular soil, one need not access all the nodes, which donot have information related to that particular soil, thus reducing the number of nodes to be accessed for that particular soil. So, the actual time consumption will be less than 'n' time units. This is a tremendous improvement over R* tree and level linked R* tree. The attribute-array occupies only size of int * number of attributes (in this case, 99 attributes), and one additional pointer is needed for the links between the nodes. A split in the tree will need two pointer assignments, and so the insertion routine have a very negligent increase in time complexity when compared to R* tree. But, the overall performance of the tree in terms of time complexity will override the memory requirements. Range query is done using the basic R* tree structure only. Hence, both the range query and the point query can be handled efficiently in terms of time complexity.

Developing a Graphical User Interface for this particular application dealing with the soils of the Grady county in Oklahoma is further discussed in Chapter IV.

CHAPTER IV

DEVELOPMENT OF GUI USING VISUAL C++

Visual representation of the data stored in these structures forms a good interaction between the users and the data base. The modified index structure provides an efficient query of the data. The present application deals with the visual representation of the querying of soils in Grady County of Oklahoma.

The data is stored in the leaf nodes of the modified R* tree structure and the visual representation of this data needs accessing the leaf nodes. Leaf level multi-linked R* tree is used because of the ease with which the data can be graphically shown as well as the basic R* tree for spatial querying.

The application consists of

1. Modifying the existing index structure.
2. Building a GUI for ease of use of querying.

In Chapter III, the modifications made to the existing structure were discussed which lead to good access time and no duplication of data. To graphically represent the data, MS Windows are used as an interface, since they are flexible, easy to use and has user friendly tools such as menus, push buttons, list boxes, etc. The data stored in leaf level multi-linked R* tree is visualized through this interface. When querying for a single soil is done through the interface, the actual query is done through the sequential search of leaf nodes. When querying for soils within a particular area, is done through the interface, the spatial searching capabilities of R* tree is utilized.

Data Description

The data for this particular application is obtained from Dr. Mark Gregory, Department of Agriculture, OSU, Stillwater, OK. The data is pertained to the Grady County of Oklahoma state, which covers an area of 4 hectares. The area is divided into small rectangles of 200 X 200 meters. The data file itself occupies 250 kb of memory.

Data Analysis

The obtained data is analyzed and is placed in two separate files. The first file has the co-ordinates of the rectangle which holds the attribute of the soil. For example., X1, Y1, X2, Y2, ATT -- is the format in which the data is placed, where X1, Y1 represent the left top coordinates of the rectangle and X2, Y2 represent the right bottom coordinates of the rectangle and ATT is the attribute that represents the soil that is present in that rectangle. For each of these attributes, there is a detail description of the soil and the number of rectangles the soil occupies. This is shown in Appendix A. The first column represents the attribute value, the next column gives the detailed description of the soil and the third column gives the number of rectangles of size 200 X 200 meters, in which the soil is spread across. The attribute codes are stored in the leaf nodes of the R* tree and the attribute linked-list is traversed based on the connecting links thus making the sequential access quick.

A look into Visual C++

With MS Windows being the interface for this particular application, Visual C++ package is used for building the interface. The product has good credibility because it comes from Microsoft,, the author of Windows itself, and also the package contains the most powerful Windows based application framework. Microsoft Foundation Class Library version 2.0 is the core of the application framework consisting of a library of C++

classes and global functions along with source code. Visual C++ also has other tools such as Visual Workbench, App Studio, the compiler, the linker, and AppWizard, ClassWizard, to construct the applications.

Microsoft Visual C++ is one of the tools for building and debugging applications that are developed in Windows environment. It incorporates high-level C++ application framework classes and integrated Windows-hosted development tools thus making the complex job of developing applications for Windows, easy.

Microsoft Visual C++ has the ability to create OLE custom controls as well as portability to different platforms. Visual C++ 2.0's 32 bit MFC library is source code-portable on to different platforms such as Intel, MIPS RISC 4000, Mac, DEC Alpha, etc. The OLE Custom Control Development Kit generates extensive code for the programmers and the programmers has to add only the code specific to the application requirements [Nanc, 94].

While developing a new application, AppWizard is first used to create the C++ Microsoft Foundation Class Library source files for the project. AppStudio is used to create and edit resources such as menus, toolbars, dialog boxes, etc. ClassWizard is used to add C++ framework code for classes and message maps for Document and View classes or resource classes [Krug, 93].

Visual Workbench:

Visual Workbench generates a make file called 'project file' with an extension of MAK. A project is a collection of interrelated source files that are compiled, linked and made into a working Windows program. Project source files are generally stored in a separate subdirectory. A project also includes the 'include files' and 'library files'. After a project is created, source code files can be edited in individual windows. All the compiler and linker switch settings can be saved through dialog boxes and to generate an executable, the Build command from the Visual Workbench Project menu is chosen.

Visual Workbench has a useful text editor that follows Windows interface standards and highlights C++ syntax with color.

App Studio:

App Studio is used to edit resources such as dialog boxes, bitmaps, and fonts. App Studio includes both a WYSIWYG menu editor and a powerful dialog box editor. In addition to handling dialog boxes, icons, cursors and bitmaps, it also edits string tables and accelerator tables. For example, to modify a dialog box -- the size of the window can be decreased or enlarged by dragging the right and the bottom borders or the OK button can be moved around or the text can be changed, etc [Krug, 93].

AppWizard:

AppWizard is accessed from the Visual Workbench Project Menu. AppWizard generates all the source files and the resource files required for the application. Every Visual C++ application starts from here. After specifying the project name, AppWizard creates a subdirectory for the project and also for the related source files and header files. By selecting options in AppWizard, all skeleton source files with differing levels of functionality are created. An AppWizard generated application with all the options has:

- A single/multiple document interface (SDI/MDI).
- Menus and dialog boxes for opening and saving files, print preview, and printing.
- Support for object linking and embedding (OLE).
- Support for Microsoft Visual Basic™ custom controls (custom VBX controls).
- Support for Help.
- A functional toolbar and status bar.

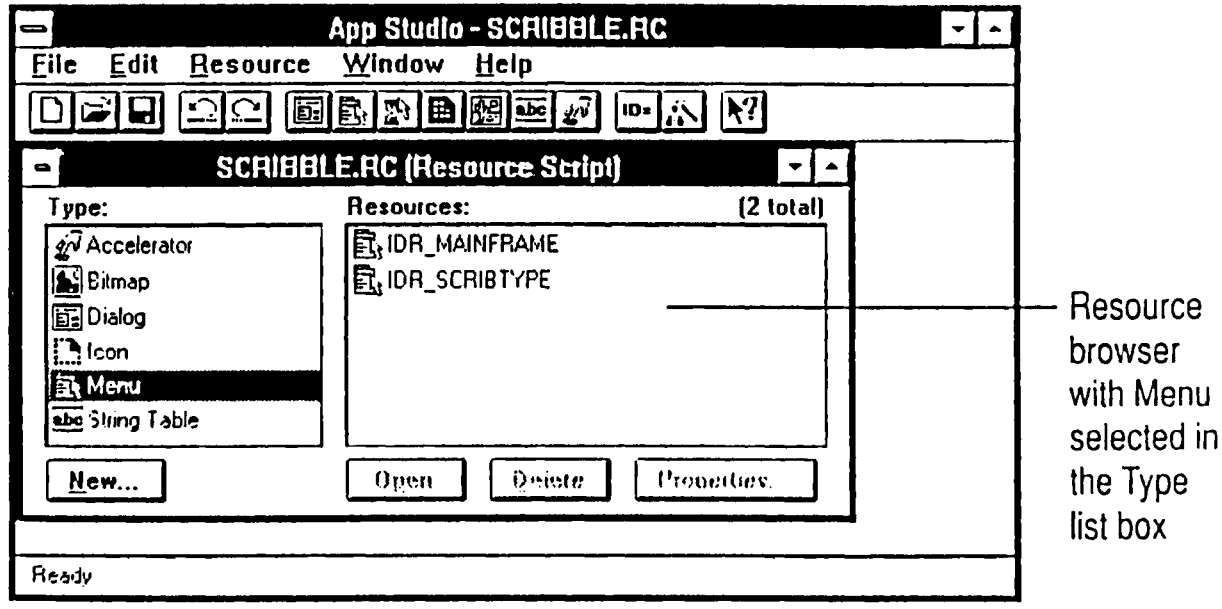


Figure 5 The App Studio Resource Browser

(Source: [MSVC, 93])

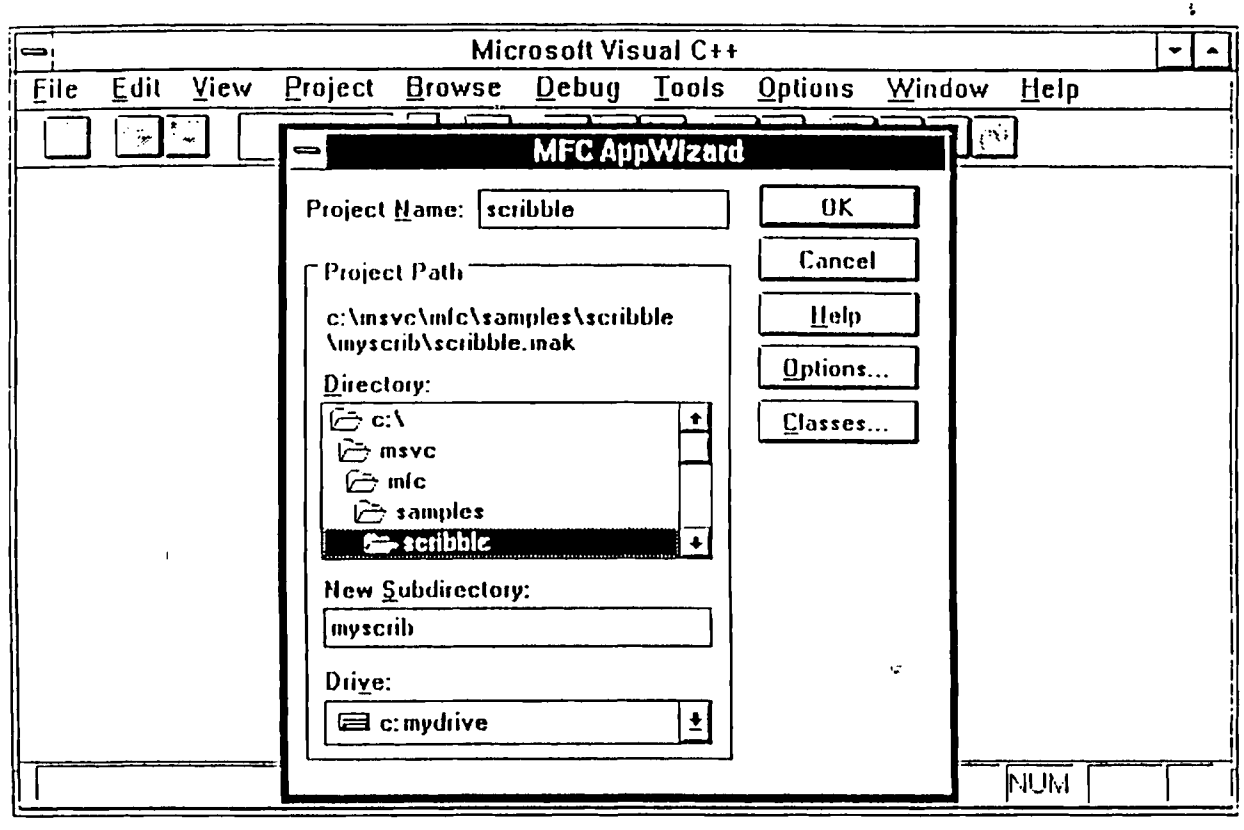


Figure 6a AppWizard Dialog Box (Source: [MSVC, 93])

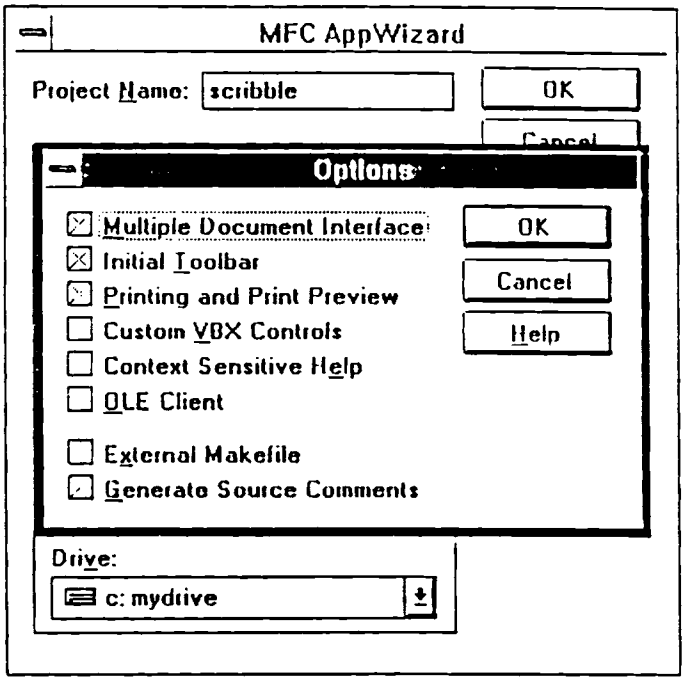


Figure 6b AppWizard Options Dialog Box

(Source: [MSVC, 93])

ClassWizard:

ClassWizard creates the necessary source files, declaration code and implementation code to derive a new class.

ClassWizard is a tool that helps in

- Creating new classes.
- Mapping messages to class-member functions.
- Mapping controls to class-member variables.

ClassWizard is used to "bind" interface objects to code. After using AppStudio to create user-interface objects, ClassWizard is used to create member functions and message maps to handle messages from these objects.

For example, if a new item called 'Test" is added to the Edit menu in App Studio and then Class Wizard, is opened , one can select the class to which the message-handler function for that object added, select the resource identifier for the Test menu item (ID_EDIT_TEST in the Object ID list), and specify that it is a COMMAND in the Messages list, and choose the Add Function button. The Class Wizard presents a message box to alter the function name, then inserts the message-map entry, function prototype, and skeleton function code.

The C/C++ compiler compiles both C source code and C++ source code. It recognizes the language from the source code filename extension. A C source code will have an extension of C, while C++ source code has an extension of CPP or CXX. AppWizard creates a working skeleton of a Windows application with class names and source code file names that are specified through dialog boxes. ClassWizard operates inside the Visual Workbench and the App Studio. Class Wizard writes the prototypes, function bodies, and code to connect the messages to the application framework. ClassWizard adds message handlers to the classes, which means, if a Windows message

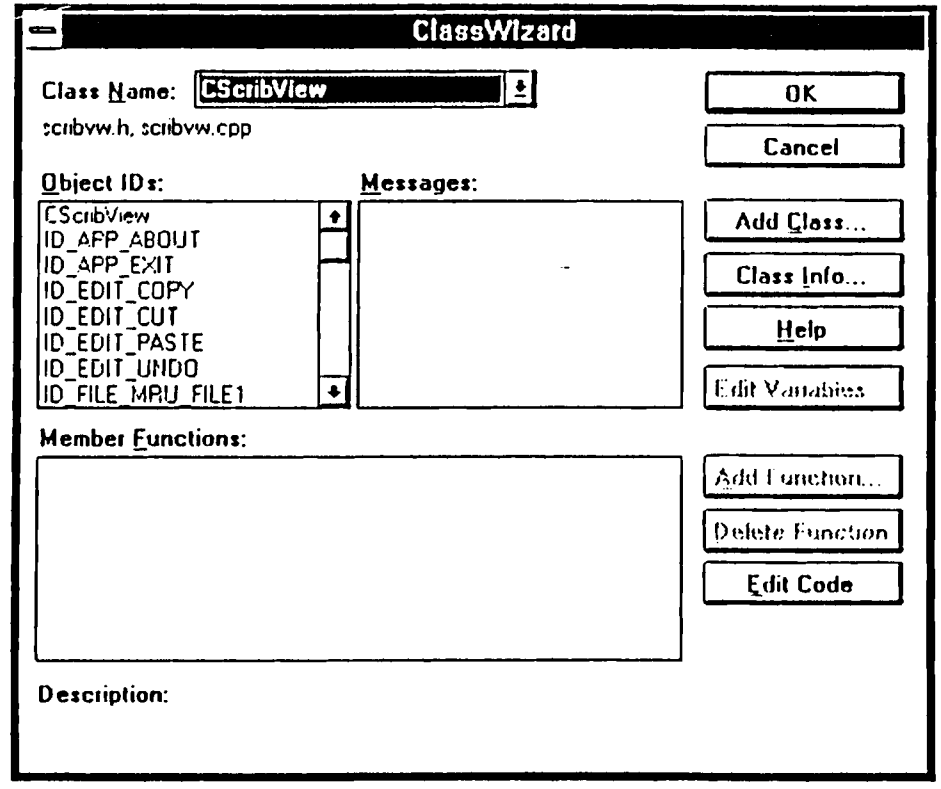


Figure 7 The Main ClassWizard Dialog Box

(Source: [MSVC, 93])

ID is selected from a list box, the Wizard generates the code with the correct function parameters and return values [Krug,93].

Documents and Views:

The 'document-view architecture' originated in 1980s in the academic area and was later used by Apple Computer in 1985 for the MacApp Application framework product. This architecture separates data from the user's view of the data, so there can be multiple views of the same data [Krug, 93]. A document is a unit of data that the user works. The document maintains, loads and stores its data. The user interacts with a documents through a "view' on the document. A view is defined as a window present in the client area of a frame window. It displays the data from it's document and takes keyboard and mouse input, which it translates them into selection and editing actions. Objects in the interface such as buttons, menus send commands to the documents, views, and other objects in the application, which carry out the commands. When the data is modified through the view, the view notifies the document. The document then sends a message to all the views (if it has multiple views) to display the new information. In the Microsoft Foundation Class Library, the base class for documents is **CDocument** and **CView** is the base class for views. In a single document interface (SDI), the view fills the main frame window, while in multiple document interface (MDI), the document frame window is displayed in the main frame window.

The main task of the framework is defining the application data in its document class(es). In this particular application, the 'document' holds the data from the leaf level multi-linked R* tree. The framework also defines how the user views and interacts the data inside the window. The framework also connects menus, buttons and other user-interface objects to commands and then defines handler function to carry the commands.

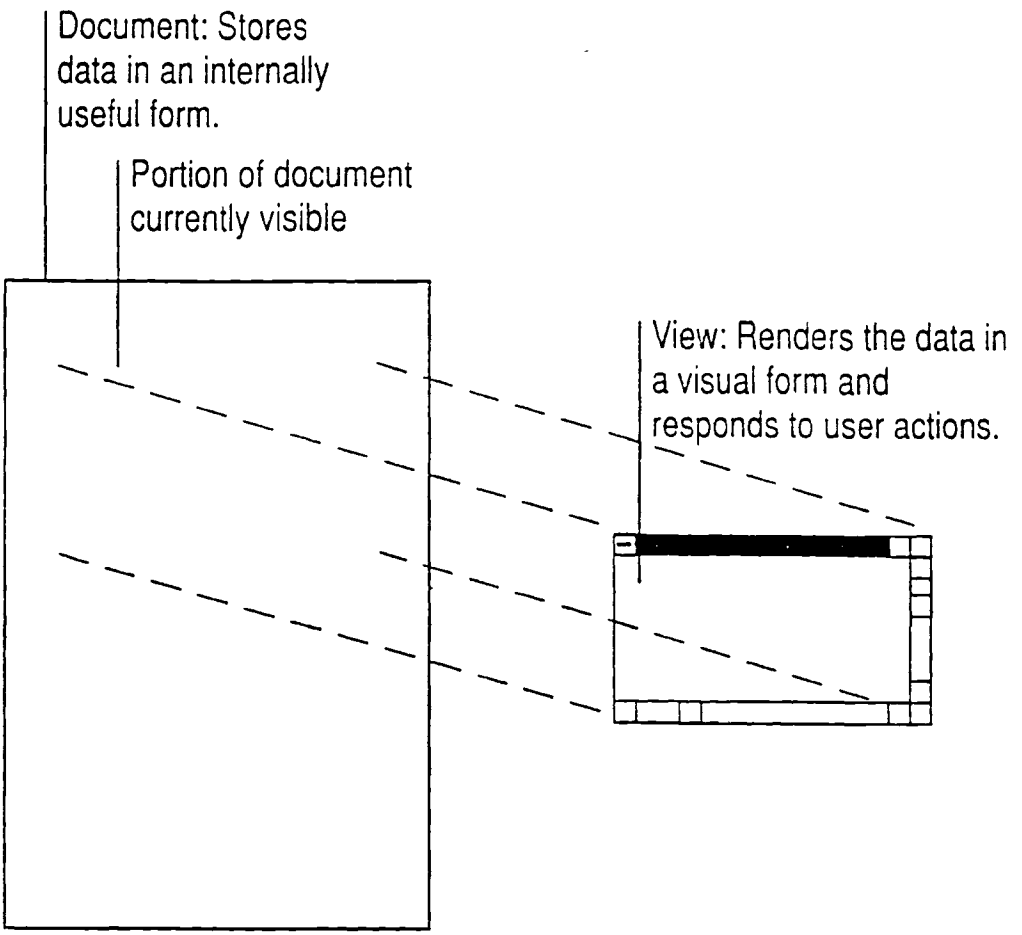


Figure 8 Document and View

(Source: [MSVC, 93])

Graphical User Interface using Visual C++

The GUI developed for this particular application picks up data from the R* tree. The main window has a menu bar with three items and each item in the menu bar has a pull down menu. The three items are File, Query, and Help. This is shown in fig. 5.

Once the user clicks on File, it brings a pull down menu with the options of Counties and Exit. Clicking on Counties will open another window titled 'Counties' and with a scrolled list of all the Counties of Oklahoma. Since only the data related to Grady County is available, selecting any other County and clicking on 'Load' button will yield an error message " Data is not available for this County for the present time". This is shown in fig. 6.

Selecting 'Grady' County and clicking on 'Load' button will open another window displaying a message "Loading the data...". The R* tree is being built by the insertion routine then, picking the data from the Grady County data file. Since the data is very huge, building the tree will take some time. After the data is loaded, if the user tries to attempt to load the data, a message is displayed regarding the status of the data being loaded as " Data already loaded for this County'. This is shown in fig. 7.

Selecting the option of 'Exit' will take the user out of the application and back into Visual C++ main window.

Now, if the user wants to Query the soils, selecting Query from the menu bar will unfold a pull-down menu, with the options of 'View Soils' and 'Query Area'. Clicking on 'View Soils' will open another window titled 'Query Soils'. This is shown in fig. 8. This window has a scrolled list, and three buttons. Since there are 99 soils spread in the County and each soil has an attribute value and it is not possible to show all the soils, a scrolled list view is provided. The window also has 3 buttons -- 'Show one', 'Show all', and 'Cancel'. Selecting a particular soil and clicking on 'Show one' will display the window titled 'Soils' drawing the outline of Grady County and the distribution of the soil

in the color that is assigned to that particular soil (fig. 9). This window has two buttons, one for the details of the color code used and the other to close the window. Clicking on the 'Details' button will give the details of the color that is assigned to that particular soil.

The 'Show all' button will display the distribution of all the soils present in the Grady County, in different colors in the window titled 'Soils' (fig. 10). Clicking on the 'Details' button will show all the color codes that are assigned to the soils.

Going back to the main window, and selecting 'Query' on the tool bar, and then selecting 'Query Area', will open another window titled 'Query Area' and displays the outline of Grady County and an empty scrolled list (fig. 11). Selecting a particular rectangular area will show all the soils in that scrolled list along with the percentage, it occupies. This is shown in fig. 12.

Displaying one soil or all the soils can be done as many times as the user wishes. Querying a particular area in the Grady County, can also be done as many times as the user desires. But the GUI doesnot allow the user to reload the data of the County, since the R* tree, which actually stores the data, has been built already.

Displaying one soil or all soils is done by picking up the data from the leaf level multi-linked list designed and developed in this thesis. Querying an area is done through the basic search routine of R tree. The 'document' actually accesses the data in the tree and displays on the 'view', while the user interacts with the data from 'view' by selecting the options.

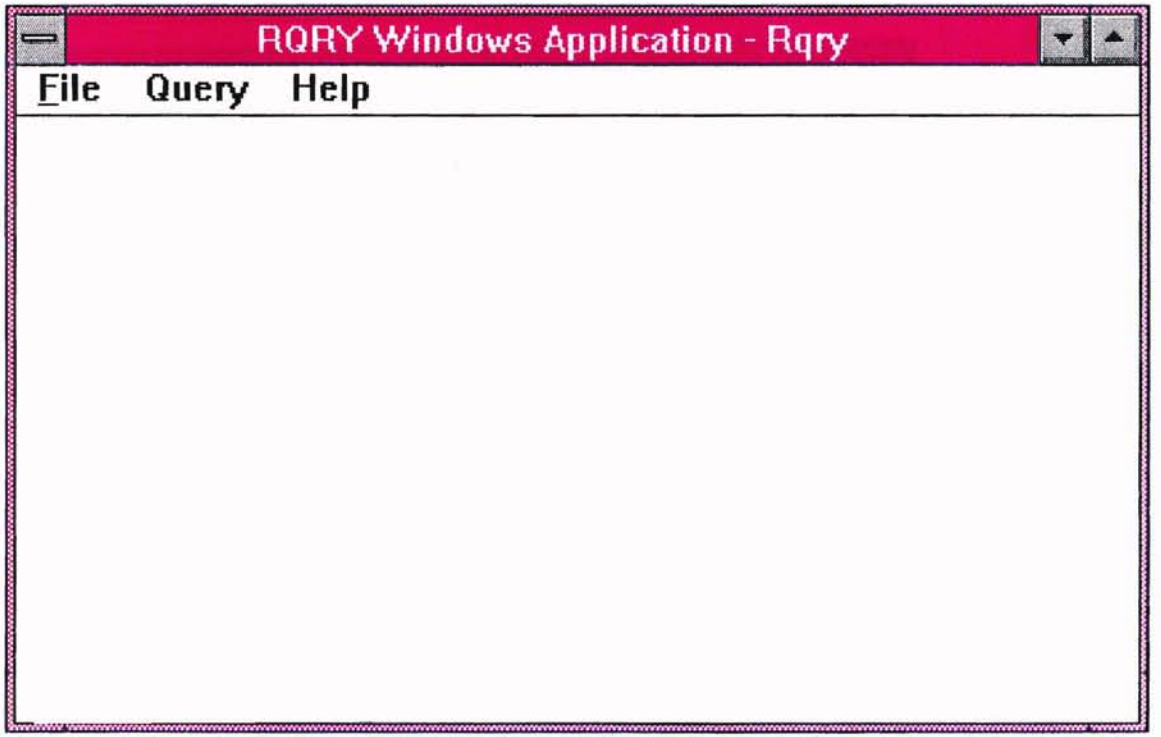


Fig 9. The main Window of the Application for Querying soils

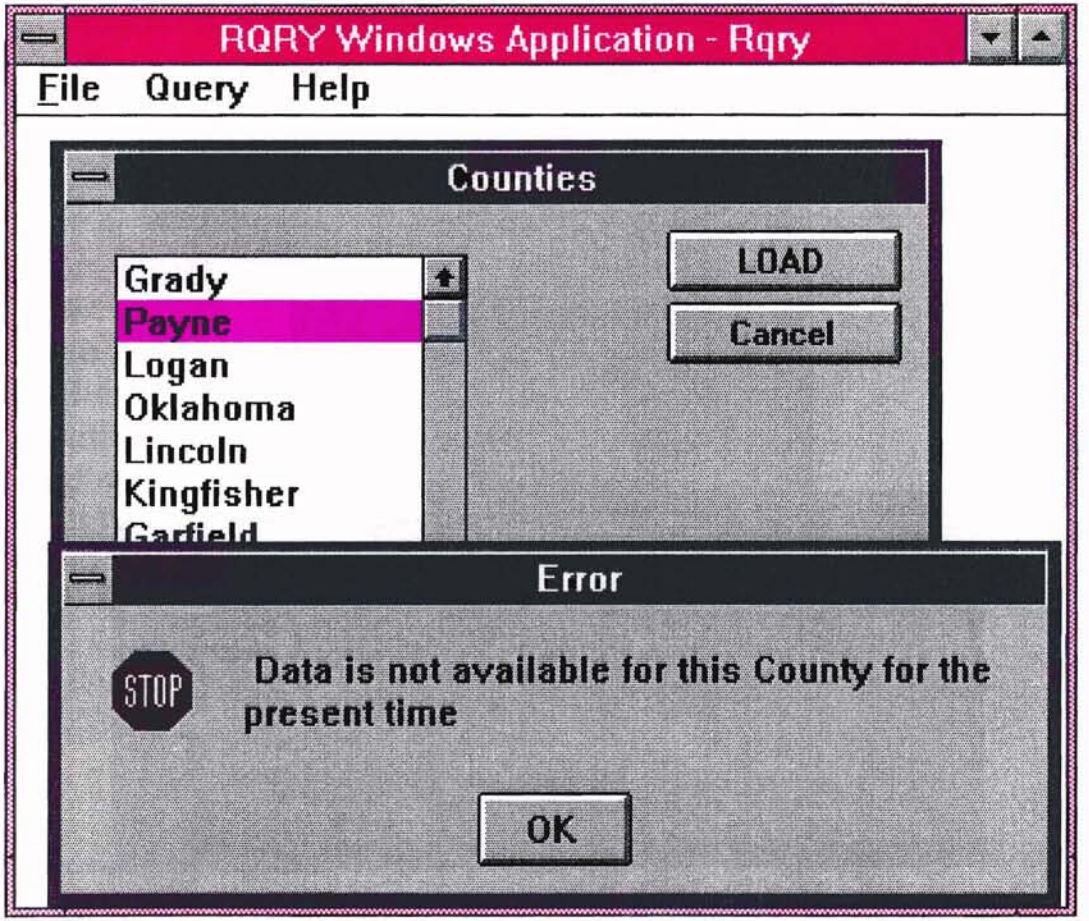


Fig 10. Popup message, when trying to load data for other Counties

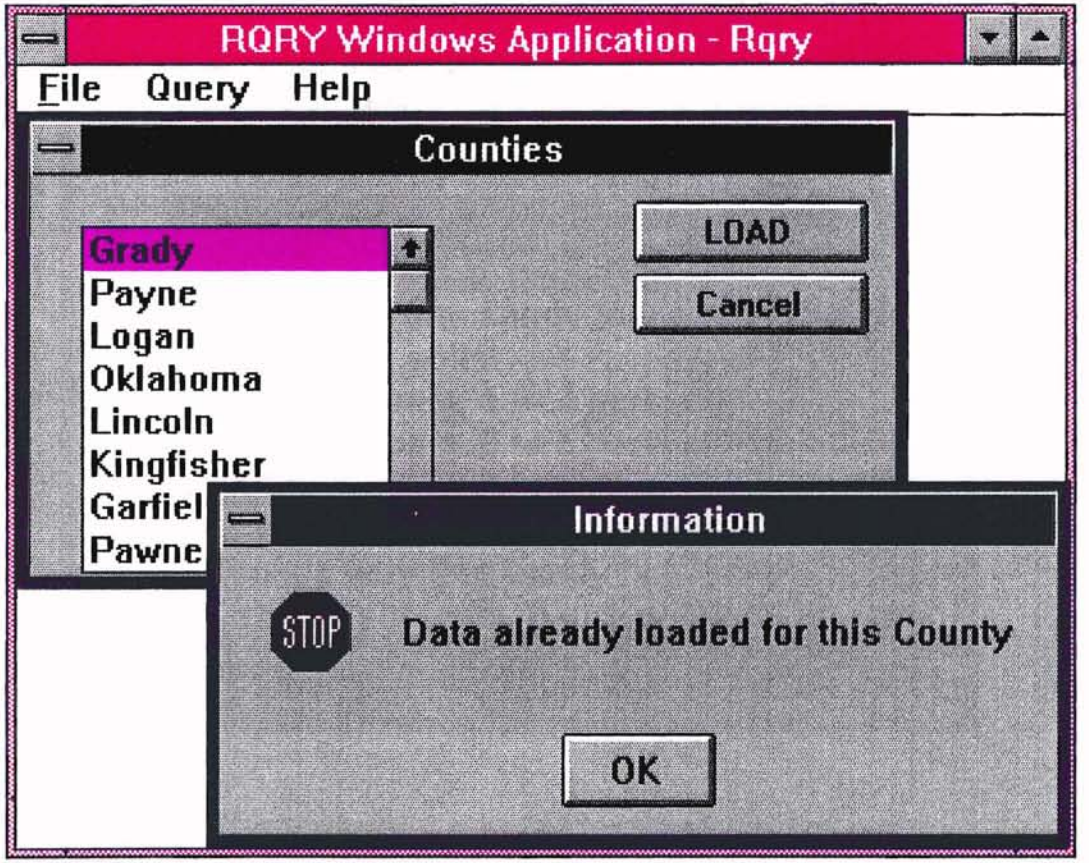


Fig 11. Message when trying to reload the data of Grady County

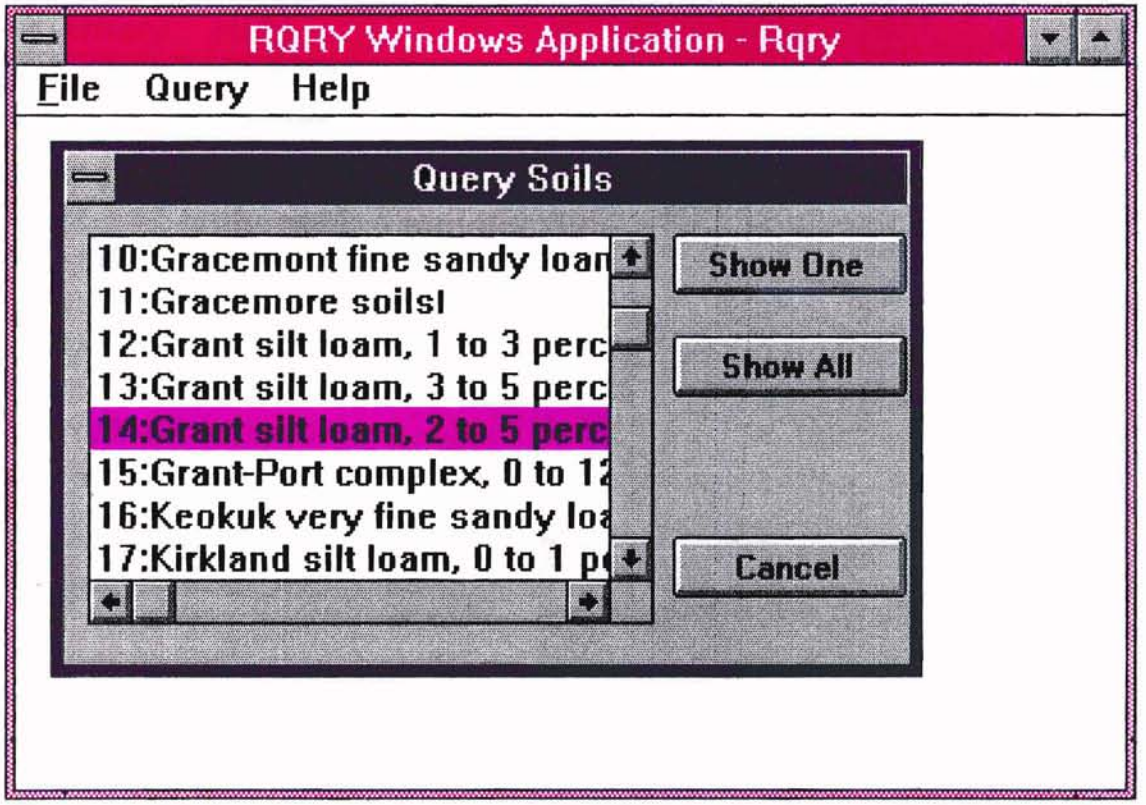


Fig 12. Query soils Window showing the names in scrolled list

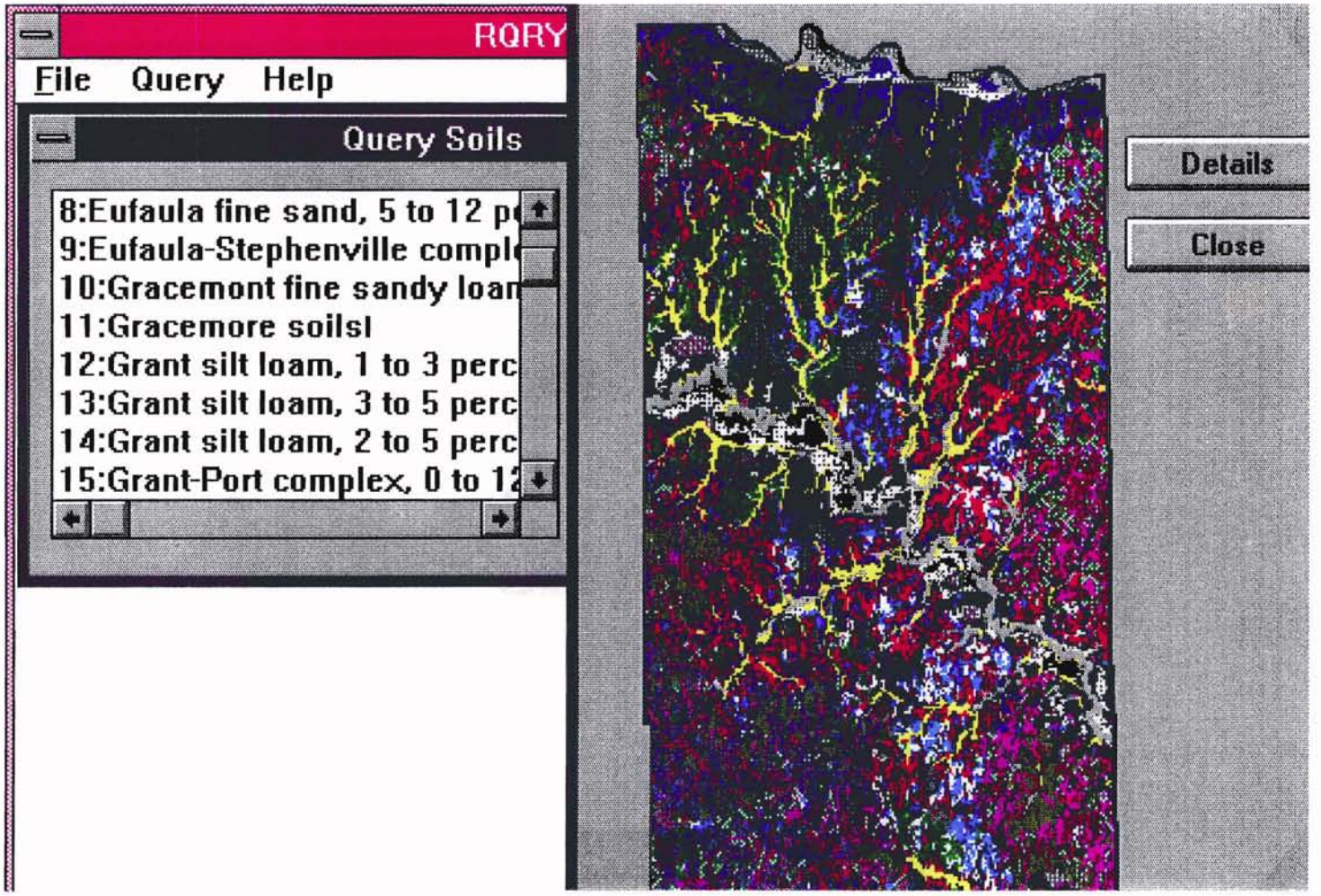


Fig 13. On clicking 'Show all', the display of all soils in Grady County

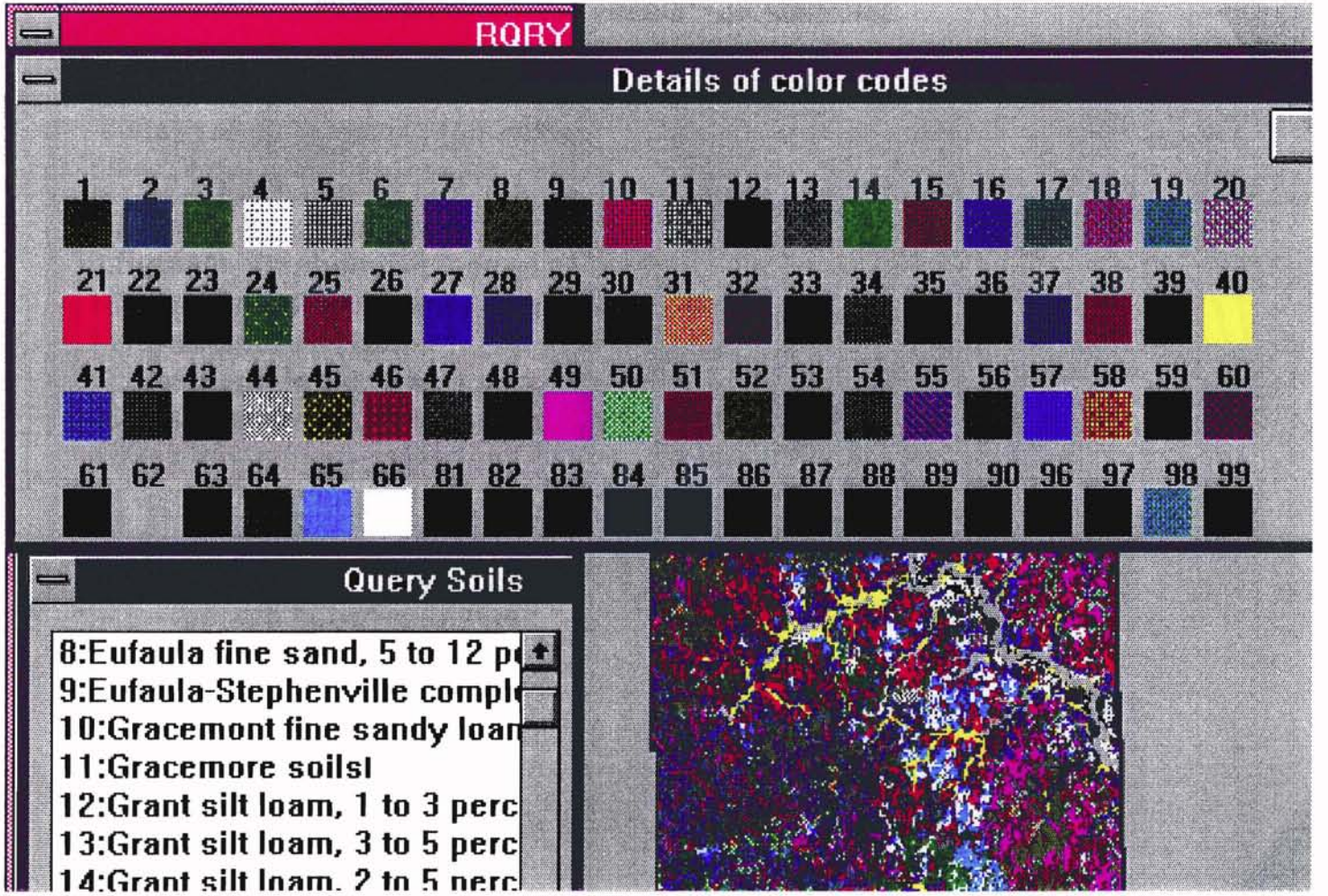


Fig 14. On clicking 'Details', displays the color codes of all soils

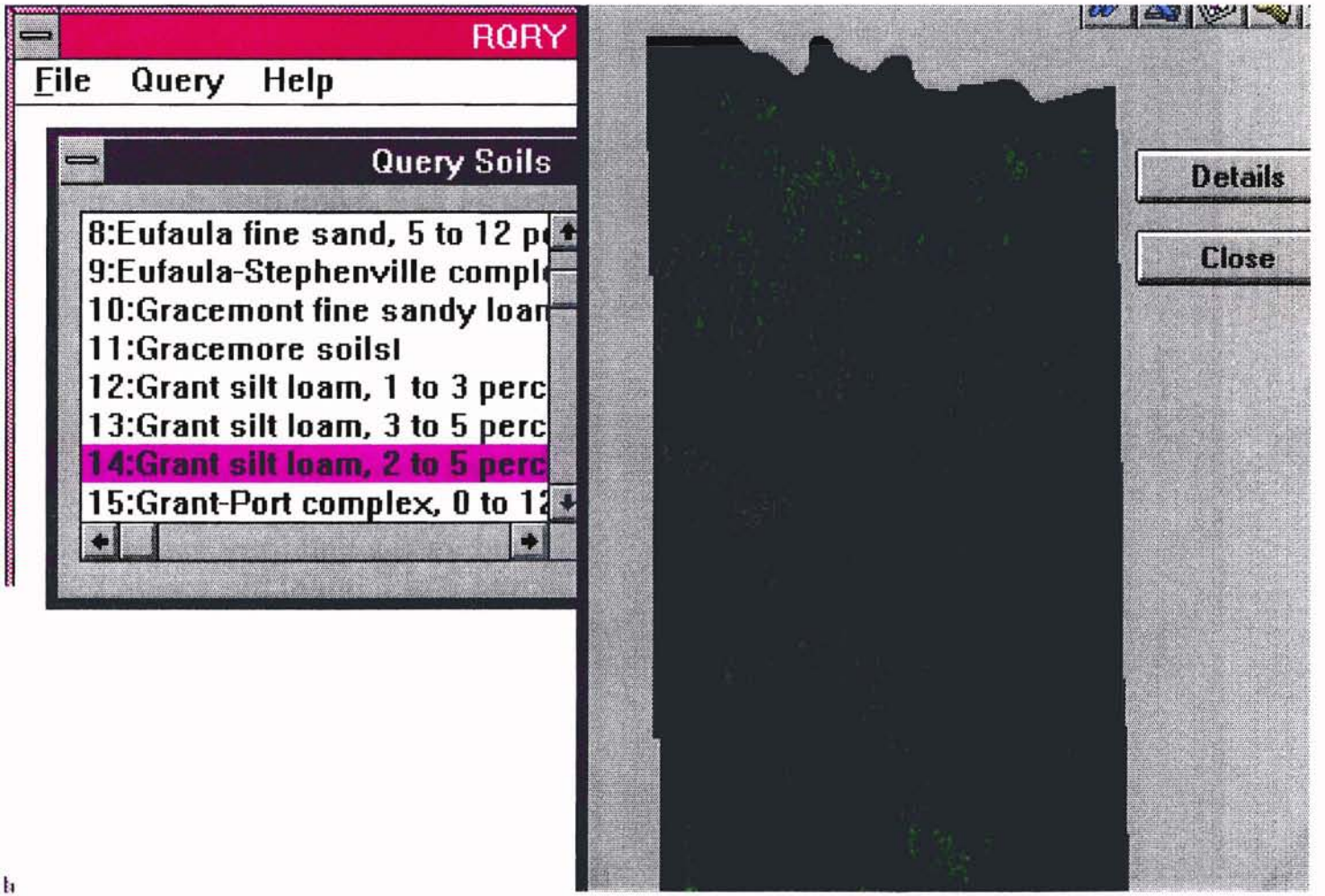


Fig 15. Display of soil with attribute 14 in color

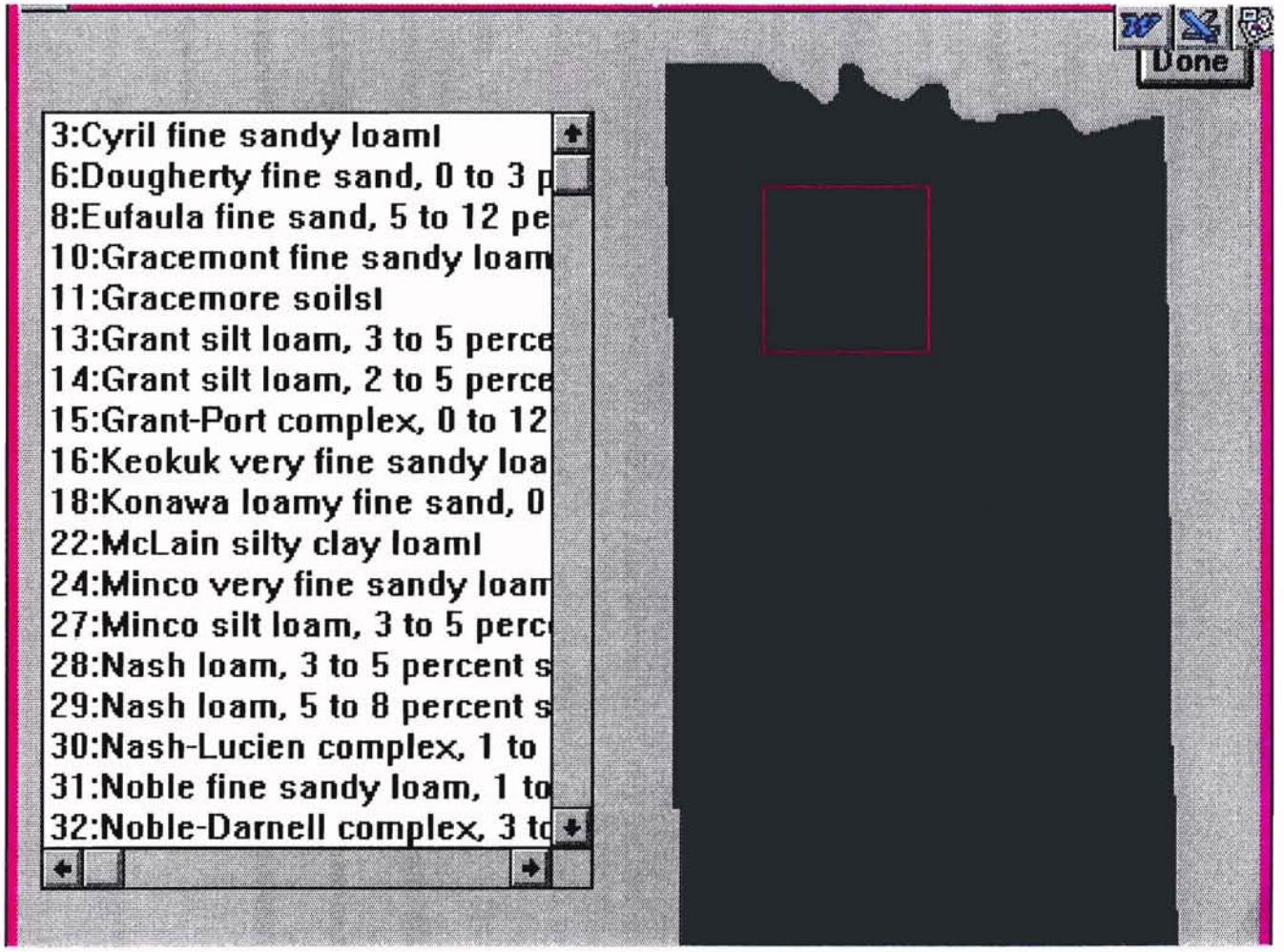


Fig 16. Selecting an area in the County, displays the names of soils in scrolled list

CHAPTER V

SUMMARY AND CONCLUSIONS

A spatial database was built for the data related to the soils of Grady County in Oklahoma. The basic R* tree structure was modified at the leaf level, by placing links to nodes that carried information related to the same soil. Different kinds of data structures were discussed, and finally a modified R* tree index structure was found to be suitable to store and retrieve the data. It was found that the leaf level multi-linked R* tree, which was designed and developed, to store the data related to the Grady County of Oklahoma, significantly reduces the time that is required to access the data sequentially. By making it multi-linked at the leaf level, to accesses data sequentially, only the leaf nodes having a particular required data can be accessed, thus reducing the time consumed, to access the data. With, data occupying about 250 K bytes of memory, this design of the R* tree is a good improvement. The cost of insertion routine for the R* tree structure that is designed and developed is very negligibly low. The newly developed R* tree has the same capabilities of R* tree for the range query, and has an improved performance for sequential search and graphical interpretation of the data.

In addition to storing this data, the visual representation of the data offered a high level of interaction between the user and the data base. Microsoft Windows provided a good interface with the powerful graphics tools that are provided by Visual C++. The

interface provides the user, the choice of selecting any county to load its data. But, in this thesis, it is restricted to only Grady County, because the data is provided for this particular County from the Agricultural department. It lets the user to query a particular soil, and displays the soil distribution in the County, in a color that is assigned to it. The user can also view all the soils that are present in the County, in different colors. It also lets the user to select a particular area in the Grady County, and shows the soils present in that area in a scrolled list.

The Graphical User Interface can be comfortably used to display the data that is stored in any spatial index structure, or their modifications. This is because of the object oriented nature of the GUI. The *Document* is the bridge between the user's view and the database. It is the documents responsibility to retrieve the data from the database and display it on the *View*. Hence the designed GUI can be used for spatial data stored in the spatial index structures. Future improvement and research can include enlarging of Query Area in the County, and displaying not only the names of soils, but also their distribution in that particular area, which will be very useful for the Agricultural department. The GUI developed in this thesis, is another aid to cut down the gap between the user and the database and each operation is user friendly, even for a novice.

REFERENCES

[Bent, 75b] Bentley, J.L. : " Multidimensional Binary Search Trees used for Associative Searching", Communications of ACM, Vol. 18, No. 9, September 1975, pp. 509 - 517.

[BF, 79b] Bentley, J.L., Friedman, J.H. : "Data Structures for Range Searching", ACM Computing Surveys, Vol. 11, No. 4, December 1992, pp. 397 - 409.

[Carl, 92] Carl, F. : " An Introduction to Geographic Information Systems: Linking Maps to Databases", Database, April 1992, pp.12 - 21.

[Cart, 92] Carter, E. : "The Evolution of C++ Programmer", Computer Language, August 1992, pp. 53 - 67.

[CTN, 87] Christos, F., Timos, S., Nick, R. : "Analysis of Object Oriented Spatial Access Methods", Proc. of the ACM SIGMOD Int. Conf., on Management of Data, 1987, pp.426 - 439.

[FE, 92] Frank, A.U., Egenhofer, M.J. : "Computer Cartography for GIS : An Object Oriented View on the Display Transformation", Computers and Geosciences, Vol. 18, No. 8, 1992, pp. 975 - 987.

[Fran, 92] Frank, A. U. : " Spatial concepts, geometric data models, and geometric data structures", Computers and Geosciences Vol. 18, No. 4, 1992, pp. 409 - 417.

[Good, 92] Goodchild, M. F. : " Geographical Data Modelling", Computers and Geosciences, Vol. 18, No. 4, 1992, pp. 401 - 408.

[Greg, 91] Greg Voss : " Object-Oriented Programming : An Introduction", Osborne McGraw-Hill, Reading, 1991.

[Gutt, 84] Guttman, A. : "R -Trees : A Dynamic Index Structure for Spatial Searching", Proc. of the ACM SIGMOD Int. Conf., on Management of Data, 1984, pp. 47 - 57.

[HS, 92] Hoel, G., Erik, Samet Hanen : "A Qualitative Study of Data Structures for Large Line Segment Databases", ACM Sigmod, 1992, pp. 205 - 214.

[KA, 90] Khoshafian, S., Abnous, R. : Object Orientation. Concepts, Languages, Databases, User Interfaces, John Wiley & Sons. Reading, 1990.

[KD, 76] Klinger, A., Dyer, C.R. : "Experiments in Picture Representation Using Regular Decomposition", Computer graphics and Image Processing, Vol. 5, No. 1, March 1976, pp.68 - 105.

[Krug, 93] Kruglinski, D. J : "Inside Visual C++", Microsoft Press, Reading, Washington, 1993.

[Knut, 73b] Knuth, D. E. : "Sorting and Searching", The Art of Computer Programming, Vol. 3, Addison - Wesley, Reading, MA, 1973, pp. 554.

[LODL, 92] Lu, H., Ooi, B. C., D'Souza, A., Low, C. C. : " Storage management in Geographic Information Systems", Lecture notes in Computer Science, Advances in spatial databases, 2nd Symposium. SSD 1991, Zurich, Switzerland, August 1991 Proceedings, pp. 451 - 470.

[Mand, 93] Mandelkern, D. : "Graphical User Interfaces : The Next Generation", Communication of ACM, April 1993, pp. 36 - 40.

[MSVC, 93] Microsoft Visual C++, Development System for Windows, Microsoft Press, 1993.

[Nanc, 94] Nancy, N. : "Borland's and Microsoft's Latest Reflect Different Priorities", Byte, November 1994, pp. 38.

[NHRB, 90] Nobert, B., Hans, P.K., Ralf, S., Bernhand, S. : "The R* Tree : An Efficient and Robust Access method for Points and Rectangles", ACM SIGMOD, New York, Vol. 19, No. 2, 1990,.pp. 322 - 331.

[NK, 92] Nixdorf, T., Kiyooka, G. : "Substance and Style : GUI design and culture", Computer Language, February 1992, pp. 43 - 58.

[OH, 91] Orfali, R., Harkey, D. : "Get GUI", Computer Language, July 1991, pp. 36 - 66.

[Reddy, 93] V. C. S. Reddy Kummetha. : "A Level Linked R* Tree Structure With an Application Using X-Window Graphical Interface", Master's thesis, Department of Computer Science, Oklahoma State University, December, 1993.

[Robi, 81] Robinson, J.L. : "The k-d-B tree : a Search Structure for Large Multidimensional Dynamic Indexes", Proceedings of the SIGMOD Conference, Ann Arbor, MI, April 1981, pp. 10 - 18.

[Same, 90] Samet, H. : The Design and Analysis of Spatial data structures, Addison - Wesley, Reading, MA, 1990.

[SHS, 86] Stonebraker, M., Hanson, E., Sellis, T. : "Rule Indexing Implementations in Database Systems", Proceedings of the First International Conference on Expert Database Systems, Charleston, SC, October, 1986.

[Stev, 46] Stevens, S. : "On the Theory of Scales of Measurement", Science Magazine, Vol. 103, No. 2684, pp. 677 - 680.

[Vert, 91] Verts, W.T. : "Object - oriented Spatial data Structures", Remote Sensing Tech. Pap 91, ACSM ASPRS Annual convention Papers published by ACSM, Bethesda, MD, USA, pp. 455 - 462.

[Vois, 91] Voisard, A. : "Towards a Toolbox for Geographic User Interfaces", Advances in Spatial Databases, 2nd Symposium, SSD 1991; Zurich, Switzerland, August 1991 Proceedings, pp. 75 - 97.

APPENDIX - A

99 categories
Grady County OK, Soils Map - 4 Hectares (9.88 ac.)

Attribute	Name of soil	Rectangles occupied
0:	no data	10851
1:	Amber very fine sandy loam, 1 to 3 percent slopes	142
2:	Bethany silt loam, 0 to 1 percent slopes	986
3:	Cyril fine sandy loam	362
4:	Dale silt loam	1468
5:	Darnell-Noble complex, 8 to 20 percent slopes	299
6:	Dougherty fine sand, 0 to 3 percent slopes	109
7:	Dougherty-Eufaula complex, 3 to 8 percent slopes	1208
8:	Eufaula fine sand, 5 to 12 percent slopes	244
9:	Eufaula-Stephenville complex, 8 to 20 percent slopes	244
10:	Gracemont fine sandy loam	971
11:	Gracemore soils	368
12:	Grant silt loam, 1 to 3 percent slopes	946
13:	Grant silt loam, 3 to 5 percent slopes	793
14:	Grant silt loam, 2 to 5 percent slopes, eroded	3643
15:	Grant-Port complex, 0 to 12 percent slopes	1711
16:	Keokuk very fine sandy loam	185
17:	Kirkland silt loam, 0 to 1 percent slopes	911
18:	Konawa loamy fine sand, 0 to 3 percent slopes	1171
19:	Konawa-Stephenville complex, 2 to 8% slopes, severely eroded	953
20:	Lela silty clay	205
21:	Lucien-Nash complex, 5 to 12 percent slopes	7380
22:	McLain silty clay loam	758
23:	Minco very fine sandy loam, 5 to 8 percent slopes	309
24:	Minco very fine sandy loam, 8 to 30 percent slopes	171
25:	Minco silt loam, 0 to 1 percent slopes	186
26:	Minco silt loam, 1 to 3 percent slopes	1283
27:	Minco silt loam, 3 to 5 percent slopes	2410
28:	Nash loam, 3 to 5 percent slopes	753
29:	Nash loam, 5 to 8 percent slopes	831
30:	Nash-Lucien complex, 1 to 5 percent slopes	2367

Attribute	Name of soil	Rectangles occupied
31:	Noble fine sandy loam, 1 to 3 percent slopes	427
32:	Noble-Darnell complex, 3 to 5 percent slopes	902
33:	Norge silt loam, 0 to 1 percent slopes	353
34:	Norge silt loam, 1 to 3 percent slopes	1287
35:	Norge silt loam, 2 to 5 percent slopes, eroded	486
36:	Pocasset silty clay loam	253
37:	Pond Creek silt loam, 0 to 1 percent slopes	256
38:	Pond Creek silt loam, 1 to 3 percent slopes	440
39:	Port fine sandy loam, overwash	1104
40:	Port silt loam	3584
41:	Pulaski fine sandy loam	464
42:	Quinlan-Rock outcrop complex, 12 to 30 percent slopes	694
43:	Reinach silt loam	860
44:	Renfrow silt loam, 1 to 3 percent slopes	1264
45:	Renfrow silt loam, 2 to 5 percent slopes, eroded	1490
46:	Renfrow silt loam, 2 to 5 percent slopes, serverely eroded	501
47:	Renfrow-Hinkle complex, 1 to 3 percent slopes	183
48:	Stephenville fine sandy loam, 1 to 3 percent slopes	689
49:	Stephenville fine sandy loam, 3 to 5 percent slopes	2441
50:	Stephenville fine sandy loam, 2 to 5 percent slopes, eroded	2358
51:	Stephenville fine sandy loam, 2 to 8% slopes,severely eroded	1908
52:	Stephenville-Darnell complex, 1 to 3 percent slopes	2086
53:	Stephenville-Eufaula complex, 3 to 8 percent slopes	2062
54:	Stephenville-Pulaski complex, 0 to 12 percent slopes	434
55:	Teller loam, 1 to 3 percent slopes	624
56:	Teller loam, 3 to 5 percent slopes	294
57:	Teller loam, 2 to 5 percent slopes, eroded	1338
58:	Teller loam, 5 to 8 percent slopes	197
59:	Tivoli loamy fine sand	39
60:	Windthorst fine sandy loam, 1 to 3 percent slopes	465
61:	Windthorst fine sandy loam, 2 to 5 percent slopes, eroded	194
62:	Yahola fine sandy loam	2279
63:	Zaneis loam, 1 to 3 percent slopes	619
64:	Zaneis loam, 3 to 5 percent slopes	313
65:	Zaneis loam, 2 to 5 percent slopes, eroded	3022
66:	Zaneis loam, 2 to 8 percent slopes, severely eroded	2108
81:	Fill	0
82:	Borrow Pits	0
83:	Gravel Pits	0
84:	Mine Pits and Dumps	0
85:	Oil-Waste Land	0
86:	Pits	0
87:	Pits, Quarries	0

Attribute	Name of soil	Rectangles occupied
88:	Quarries	0
89:	Slickspot	0
90:	Strip Mines	0
96:	Water, Sand Channel	126
97:	Urban	0
98:	Water	200
99:	Border	1246

VITA

Sujatha S. Neelam

Candidate for the Degree of

Master of Science

Thesis: DESIGN AND IMPLEMENTATION OF AN EFFICIENT INDEX STRUCUTURE AND A GUI FOR SPATIAL DATABASES USING VISUAL C++

Major Field: Computer Science

Biographical:

Education: Graduated from Sri Satya Sai Institute of Higher Learning, Anantapur, AP, India, in 1982. Received Bachelor of Technology degree in Electrical and Electronics Engineering from Sri Venkateswara University, Tirupati, AP, India, in May 1986. Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in May 1995.

Experience: Worked as HelpDesk Consultant at the HelpDesk, University Computer Center, Oklahoma State University, 1992 to present.