The Design and Analysis of Spatial Data Structures

Unified Patents Exhibit 1005 App'x A-N

The Design and Analysis of Spatial Data Structures

Hanan Samet
UNIVERSITY OF MARYLAND

ADDISON - WESLEY PUBLISHING COMPANY, INC.
Reading, Massachusetts • Menlo Park, California • New York
Don Mills, Ontario • Wokingham, England • Amsterdam
Bonn • Sydney • Singapore • Tokyo • Madrid • San Juan

Unified Patents Exhibit 1005 App'x A-N

This book is in the Addison-Wesley Series in Computer Science
Michael A. Harrison: Consulting Editor

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Addison-Wesley was aware of a trademark claim, the designations have been printed in initial caps or all caps.

The programs and applications presented in this book have been included for their instructional value. They have been tested with care, but are not guaranteed for any particular purpose. The publisher does not offer any warranties or representations, nor does it accept any liabilities with respect to the programs or applications.

Library of Congress Cataloging-in-Publication Data

Samet, Hanan.
The Design and analysis of spatial data structures/by Hanan Samet.
p. cm.
Bibliography: p.
Includes index.
ISBN 0-201-50255-0
1. Data structures (Computer science) 2. Computer graphics.
I. Title.
QA76.9.D35S26 1989 89-30382
005.7'3—dc19 CIP

Reprinted with corrections January, 1994

Copyright © 1990 by Addison-Wesley Publishing Company, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America. Published simultaneously in Canada.

4 5 6 7 8 9 10 11 12 13 14-MA-97 96 95 94

Credits:
Thor Bestul created the cover art.

Gyuri Fekete generated Figure 1.16; Daniel DeMenthon, Figures 1.20, 1.21, and 1.23; Jiang-Hsing Chu, Figures 2.48 and 2.52; and Walid Aref, Figures 4.38 through 4.40.

Figures 1.1, 4.9, and 4.10 are from H. Samet and R. E. Webber, On encoding boundaries with quadtrees, *IEEE Transactions on Pattern Analysis and Machine Intelligence 6*, 3 (May 1984), 365–369. © 1984 IEEE. Reprinted by permission of IEEE.

Figures 1.2, 1.3, 1.5 through 1.10, 1.12, 1.14, 1.25, 1.26, 2.3, 2.4, 2.18, 2.20, 2.30, 2.32, 2.53, 2.54, 2.57, 2.58, 3.20, 3.21, 4.1 through 4.5, 4.7, 4.8, 4.11, and 5.2 are from H. Samet, The quadtree and related hierarchical data structures, *ACM Computing Surveys 16*, 2 (June 1984), 187–260. Reprinted by permission of ACM.

Figures 1.4 and 5.6 are from H. Samet and R. E. Webber, Hierarchical data structures and algorithms for computer graphics. Part I. Fundamentals, *IEEE Computer Graphics and Applications 8*, 3 (May 1988), 48–68. © 1988 IEEE. Reprinted by permission of IEEE.

Figure 1.30 is from M. Li, W. I. Grosky, and R. Jain, Normalized quadtrees with respect to translations, *Computer Graphics and Image Processing 20*, 1 (September 1982), 72–81. Reprinted by permission of Academic Press.

Figures 2.7 and 2.10 through 2.15 are from H. Samet, Deletion in two-dimensional quad trees, *Communications of the ACM 23*, 12 (December 1980), 703–710. Reprinted by permission of ACM.

Figures 2.26 and 2.27 are from D. T. Lee and C. K. Wong, Worst-case analysis for region and partial region searches in multidimensional binary search trees and quad trees, *Acta Informatica 9*, 1 (1977), 23–29. Reprinted by permission of Springer Verlag.
Continued on p. 493

To my parents, Julius and Lotte

PREFACE

Spatial data consist of points, lines, rectangles, regions, surfaces, and volumes. The representation of such data is becoming increasingly important in applications in computer graphics, computer vision, database management systems, computer-aided design, solid modeling, robotics, geographic information systems (GIS), image processing, computational geometry, pattern recognition, and other areas. Once an application has been specified, it is common for the spatial data types to be more precise. For example, consider a geographic information system (GIS). In such a case, line data are differentiated on the basis of whether the lines are isolated (e.g., earthquake faults), elements of tree-like structures (e.g., rivers and their tributaries), or elements of networks (e.g., rail and highway systems). Similarly region data are often in the form of polygons that are isolated (e.g., lakes), adjacent (e.g., nations), or nested (e.g., contours). Clearly the variations are large.

Many of the data structures currently used to represent spatial data are hierarchical. They are based on the principle of recursive decomposition (similar to *divide and conquer* methods [Aho74]). One such data structure is the quadtree (octree in three dimensions). As we shall see, the term *quadtree* has taken on a generic meaning. In this book, it is my goal to show how a number of hierarchical data structures used in different domains are related to each other and to quadtrees. My presentation concentrates on these different representations and illustrates how a number of basic operations that use them are performed.

Hierarchical data structures are useful because of their ability to focus on the interesting subsets of the data. This focusing results in an efficient representation and in improved execution times. Thus they are particularly convenient for performing set operations. Many of the operations described can often be performed as efficiently, or more so, with other data structures. Nevertheless hierarchical data structures are attractive because of their conceptual clarity and ease of implementation. In addition, the use of some of them provides a spatial index. This is very useful in applications involving spatial databases.

As an example of the type of problems to which the techniques described in this book are applicable, consider a cartographic database consisting of a number of maps and some typical queries. The database contains a contour map, say at 50-foot elevation intervals, and a land use map classifying areas according to crop growth. Our goal is to determine all regions between 400- and 600-foot elevation levels where wheat is grown. This will require an intersection operation on the two maps. Such an analysis could be rather costly, depending on the way the maps are represented. For example, since areas where corn is grown are of no interest, we wish to spend a minimal amount of effort searching such regions. Yet traditional region representations such as the boundary code [Free74] are very local in application, making it difficult to avoid examining a corn-growing area that meets the desired elevation criterion. In contrast, hierarchical representations such as the region quadtree are more global in nature and enable the elimination of larger areas from consideration.

Another query might be to determine whether two roads intersect within a given area. We could check them point by point; however, a more efficient method of analysis would be to represent them by a hierarchical sequence of enclosing rectangles and to discover whether in fact the rectangles do overlap. If they do not, the search is terminated. If an intersection is possible, more work may have to be done, depending on which method of representation is used.

A similar query can be constructed for point data — for example, to determine all cities within 50 miles of St. Louis that have a population in excess of 20,000. Again we could check each city individually. However, using a representation that decomposes the United States into square areas having sides of length 100 miles would mean that at most four squares need to be examined. Thus California and its adjacent states can be safely ignored.

Finally, suppose we wish to integrate our queries over a database containing many different types of data (e.g., points, lines, areas). A typical query might be, "Find all cities with a population in excess of 5,000 people in wheat-growing regions within 20 miles of the Mississippi River." In this book we will present a number of different ways of representing data so that such queries and other operations can be efficiently processed.

This book is organized as follows. There is one chapter for each spatial data type, in which I present a number of different data structures. The aim is to gain the ability to evaluate them and to determine their applicability. Two problems are treated in great detail: the rectangle intersection problem, discussed in the context of the representation of collections of small rectangles (Chapter 3), and the point location problem, discussed in the context of the representation of curvilinear data (Chapter 4). A comprehensive treatment of the use of quadtrees and octrees in other applications in computer graphics, image processing, and geographic information systems (GIS) can be found in [Same90b].

Chapter 1 gives a general introduction to the principle of recursive decomposition with a concentration on two-dimensional regions. Key properties, as well as a historical overview, are presented.

Chapter 2 discusses hierarchical representations of multidimensional point data. These data structures are particularly useful in applications in database management systems because they are designed to facilitate responses to search queries.

Chapter 3 examines the hierarchical representation of collections of small rectangles. Such data arise in applications in computational geometry, very large-scale integrations (VLSI), cartography, and database management. Examples from these fields (e.g., the rectangle intersection problem) are used to illustrate their differences. Many of the representations are closely related to those used for point data. This chapter is an expansion of [Same88a].

Chapter 4 treats the hierarchical representation of curvilinear data. The primary focus is on the representation of polygonal maps. The goal is to be able to solve the point location problem. Quadtree-like solutions are compared with those from computational geometry such as the K-structure [Kirk83] and the layered dag [Edel86a].

Chapter 5 looks at the representation of three-dimensional region data. In this case, a number of octree variants are examined, as well as constructive solid geometry (CSG) and the boundary model (BRep). Algorithms are discussed for converting between some of these representations. The representation of surfaces (i.e., 2.5-dimensional data) is also briefly discussed in this chapter.

There are a number of topics for which justice requires a considerably more detailed treatment. However, due to space limitations, I have omitted a detailed discussion of them and instead refer interested readers to the appropriate literature. For example, surface representations are discussed briefly with three-dimensional data in Chapter 5 (also see Chapter 7 of [Same90b]). The notion of a pyramid is presented only at a cursory level in Chapter 1 so that it can be contrasted with the quadtree. In particular, the pyramid is a multiresolution representation, whereas the quadtree is a variable resolution representation. Readers are referred to Tanimoto and Klinger [Tani80] and the collection of papers edited by Rosenfeld [Rose83a] for a more comprehensive exposition on pyramids.

Results from computational geometry, although related to many of the topics covered in this book, are discussed only in the context of representations for collections of small rectangles (Chapter 3) and curvilinear data (Chapter 4). For more details on early work involving some of these and related topics, interested readers should consult the surveys by Bentley and Friedman [Bent79b], Overmars [Over88a], Edelsbrunner [Edel84], Nagy and Wagle [Nagy79], Peuquet [Peuq84], Requicha [Requ80], Srihari [Srih81], Samet and Rosenfeld [Same80d], Samet [Same84b, Same88a], Samet and Webber [Same88c, Same88d], and Toussaint [Tous80].

There are also a number of excellent texts containing material related to the topics that I cover. Rosenfeld and Kak [Rose82a] should be consulted for an encyclopedic treatment of image processing. Mäntylä [Mänt87] has written a comprehensive introduction to solid modeling. Burrough [Burr86] provides a survey of geographic information systems (GIS). Overmars [Over83] has produced a particularly good treatment of multidimensional point data. In a similar vein, see Mehlhorn's [Mehl84] unified treatment of multidimensional searching and computational geometry. For thorough introductions to computational geometry, see Preparata and

K-structure and the layered dag in Section 4.3 are relevant to computational geometry. Bucket methods such as linear hashing, spiral hashing, grid file, and EXCELL, in Section 2.8, and R-trees in Section 3.5.3 are important in the study of database management systems. Methods for multidimensional searching that are discussed include k–d trees in Section 2.4, range trees and priority search trees in Section 2.5, and point-based rectangle representations in Section 3.4. The discussions of the representation of two-dimensional regions in Chapter 1, polygonal representations in Chapter 4, and use of point methods for focussing the Hough Transform are relevant to image processing. Finally the rectangle-representation methods and plane-sweep methods of Chapter 3 are important in the field of VLSI design.

The natural home for courses that use this book is in a computer science department, but the book could also be used in a curriculum in geographic information systems (GIS). Such a course is offered in geography departments. The emphasis for a course in this area would be on the use of quadtree-like methods for representing spatial data.

Unified Patents Exhibit 1005 App'x A-N

Shamos [Prep85] and Edelsbrunner [Edel87] (also see [Prep83, ORou88]). A broader view of the literature can be found in related bibliographies such as the ongoing collective effort coordinated by Edelsbrunner [Edel83c, Edel88], and Rosenfeld's annual collection of references in the journal *Computer Vision, Graphics, and Image Processing* (e.g., [Rose88]).

Nevertheless, given the broad and rapidly expanding nature of the field, I am bound to have omitted significant concepts and references. In addition at times I devote a disproportionate amount of attention to some concepts at the expense of others. This is principally for expository purposes; I feel that it is better to understand some structures well rather than to give readers a quick runthrough of buzzwords. For these indiscretions, I beg your pardon and hope you nevertheless bear with me.

My approach is an algorithmic one. Whenever possible, I have tried to motivate critical steps in the algorithms by a liberal use of examples. I feel that it is of paramount importance for readers to see the ease with which the representations can be implemented and used. In each chapter, except for the introduction (Chapter 1), I give at least one detailed algorithm using pseudo-code so that readers can see how the ideas can be applied. The pseudo-code is a variant of the ALGOL [Naur60] programming language that has a data structuring facility incorporating pointers and record structures. Recursion is used heavily. This language has similarities to C [Kern78], PASCAL [Jens74], SAIL [Reis76], and ALGOL W [Baue68]. Its basic features are described in the Appendix. However, the actual code is not crucial to understanding the techniques, and it may be skipped on a first reading. The index indicates the page numbers where the code for each algorithm is found.

In many cases I also give an analysis of the space and time requirements of different data structures and algorithms. The analysis is usually of an asymptotic nature and is in terms of *big O* and Ω notation [Knut76]. The *big O* notation denotes an upper bound. For example, if an algorithm takes $O(\log_2 N)$ time, then its worst-case behavior is never any worse than $\log_2 N$. The Ω notation denotes a lower bound. As an example of its use, consider the problem of sorting N numbers. When we say that sorting is $\Omega(N \cdot \log_2 N)$ we mean that given any algorithm for sorting, there is some set of N input values for which the algorithm will require at least this much time.

At times I also describe implementations of some of the data structures for the purpose of comparison. In such cases counts, such as the number of fields in a record, are often given. These numbers are meant only to amplify the discussion. They are not to be taken literally, as improvements are always possible once a specific application is analyzed more carefully.

Each chapter contains a substantial number of exercises. Many of the exercises develop further the material in the text as a means of testing the reader's understanding, as well as suggesting future directions. When the exercise or its solution is not my own, I have preceded it with the name of its originator. The exercises have not been graded by difficulty. They rarely require any mathematical skills beyond the undergraduate level for their solution. However, while some of the exercises are quite straightforward, others require some ingenuity. Solutions, or references to papers that

contain the solution, are provided for a substantial number of the exercises that do not require programming. Readers are cautioned to try to solve the exercises before turning to the solutions. It is my belief that much can be learned this way (for the student and, even more so, for the author). The motivation for undertaking this task was my wonderful experience on my first encounter with the rich work on data structures by Knuth [Knut73a, Knut73b].

An extensive bibliography is provided. It contains entries for both this book and the companion text [Same90b]. Not all of the references that appear in the bibliography are cited in the two texts. They are retained for the purpose of giving readers the ability to access the entire body of literature relevant to the topics discussed in them. Each reference is annotated with a key word(s) and a list of the numbers of the sections in which it is cited in either of the texts (including exercises and solutions). In addition, a name and credit index is provided that indicates the page numbers in this book on which each author's work is cited or a credit is made.

ACKNOWLEDGMENTS

Over the years I have received help from many people, and I am extremely grateful to them. In particular Robert E. Webber, Markku Tamminen, and Michael B. Dillencourt have generously given me much of their time and have gone over critical parts of the book. I have drawn heavily on their knowledge of some of the topics covered here. I have also been extremely fortunate to work with Azriel Rosenfeld over the past ten years. His dedication and scholarship have been a true inspiration to me. I deeply cherish our association.

I was introduced to the field of spatial data structures by Gary D. Knott who asked "how to delete in point quadtrees." Azriel Rosenfeld and Charles R. Dyer provided much interaction in the initial phase of my research. Those discussions led to the discovery of the neighbor-finding principle. It is during that time that many of the basic conversion algorithms between quadtrees and other image representations were developed as well. I learned much about image processing and computer vision from them. Robert E. Webber taught me computer graphics, Markku Tamminen taught me solid modeling and representations for multiattribute data, and Michael B. Dillencourt taught me about computational geometry.

During the time that this book was written, my research was supported, in part, by the National Science Foundation, the Defense Mapping Agency, the Harry Diamond Laboratory, and the Bureau of the Census. In particular I would like to thank Richard Antony, Y. T. Chien, Su-shing Chen, Hank Cook, Phil Emmerman, Joe Rastatter, Alan Saalfeld, and Larry Tokarcik. I am appreciative of their support.

Many people helped me in the process of preparing the book for publication. Acknowledgments are due to Rene McDonald for coordinating the day-to-day matters

of getting the book out and copyediting; to Scott Carson, Emery Jou, and Jim Purtilo for TROFF assistance beyond the call of duty; to Marisa Antoy and Sergio Antoy for designing and implementing the algorithm formatter used to typeset the algorithms; to Barbara Burnett, Michael B. Dillencourt, and Sandra German for help with the index; to Jay Weber for setting up the TROFF macrofiles so that I can keep track of symbolic names and thus be able to move text around without worrying about the numbering of exercises, sections, and chapters; to Liz Allen for early TROFF help; to Nono Kusuma, Mark Stanley, and Joan Wright Hamilton for drawing the figures; to Richard Muntz and Gerald Estrin for providing temporary office space and computer access at UCLA; to Sandy German, Gwen Nelson, and Janet Salzman for help in initial typing of the manuscript; to S. S. Iyengar, Duane Marble, George Nagy, and Terry Smith who reviewed the book; and to Peter Gordon, John Remington, and Keith Wollman at Addison-Wesley Publishing Company for their encouragement and confidence in this project.

Aside from the individuals named above, I have also benefited from discussions with many other people over the past years. They have commented on various parts of the book and include Chuan-Heng Ang, Walid Aref, James Arvo, Harvey H. Atkinson, Thor Bestul, Sharat Chandran, Chiun-Hong Chien, Jiang-Hsing Chu, Leila De Floriani, Roger Eastman, Herbert Edelsbrunner, Claudio Esperanca, Christos Faloutsos, George (Gyuri) Fekete, Kikuo Fujimura, John Gannon, John Goldak, Erik Hoel, Liuqing Huang, Frederik W. Jansen, Ajay Kela, David Kirk, Per Åke Larson, Dani Lischinski, Don Meagher, David Mount, Randal C. Nelson, Glenn Pearson, Ron Sacks-Davis, Timos Sellis, Clifford A. Shaffer, Deepak Sherlekar, Li Tong, Brian Von Herzen, Peter Widmayer, and David Wise. I deeply appreciate their help.

A GUIDE TO THE INSTRUCTOR

This book can be used in a second data structures course, one with emphasis on the representation of spatial data. The focus is on the use of the principle of divide-and-conquer for which hierarchical data structures provide a good demonstration. Throughout the book both worst-case optimal methods and methods that work well in practice are emphasized in conformance with my view that the well-rounded computer scientist should be conversant with both types of algorithms. This material is more than can be covered in one semester; but the instructor can reduce it as necessary. For example, the detailed examples can be skipped or used as a basis of a term project or programming assignments.

The book can also be used to organize a course to be prerequisite to courses in computer graphics and solid modeling, computational geometry, database management systems, multidimensional searching, image processing, and VLSI design. The discussions of the representations of two-dimensional regions in Chapter 1, polygonal representations in Chapter 4, and most of Chapter 5 are relevant to computer graphics and solid modeling. The discussions of plane-sweep methods and their associated data structures such as segment trees, interval trees, and priority search trees in Sections 3.2 and 3.3 and point location and associated data structures such as the

CONTENTS

Preface **vii**

1 INTRODUCTION **1**
1.1 Basic Definitions 1
1.2 Overview of Quadtrees and Octrees 2
1.3 History of the Use of Quadtrees and Octrees 10
1.4 Space Decomposition Methods 16
1.4.1 Polygonal Tilings 17
1.4.2 Nonpolygonal Tilings 26
1.5 Space Requirements 32

2 POINT DATA **43**
2.1 Introduction 44
2.2 Nonhierarchical Data Structures 46
2.3 Point Quadtrees 48
2.3.1 Insertion 49
2.3.2 Deletion 54
2.3.3 Search 64
2.4 k-d Trees 66
2.4.1 Insertion 68
2.4.2 Deletion 73
2.4.3 Search 77
2.4.4 Comparison with Point Quadtrees 80
2.5 Range Trees and Priority Search Trees 80
2.6 Region-based Quadtrees 85
2.6.1 MX Quadtrees 86
2.6.2 PR Quadtrees 92

2.6.3 Comparison of Point and Region-based Quadtrees 104
2.7 Bit Interleaving 105
2.8 Bucket Methods 110
2.8.1 Hierarchical Bucket Methods 111
2.8.2 Nonhierarchical Bucket Methods 116
2.8.2.1 Linear Hashing 117
2.8.2.2 Spiral Hashing 125
2.8.2.3 Grid File 135
2.8.2.4 EXCELL 141
2.9 Conclusion 147

3 COLLECTIONS OF SMALL RECTANGLES 153
3.1 Introduction 155
3.2 Plane-Sweep Methods and the Rectangle Intersection Problem 158
3.2.1 Segment Trees 160
3.2.2 Interval Trees 165
3.2.3 Priority Search Trees 171
3.2.4 Alternative Solutions and Related Problems 174
3.3 Plane-Sweep Methods and the Measure Problem 178
3.4 Point-based Methods 186
3.5 Area-based Methods 199
3.5.1 MX-CIF Quadtrees 200
3.5.1.1 Insertion 202
3.5.1.2 Deletion 206
3.5.1.3 Search 209
3.5.2 Multiple Quadtree Block Representations 213
3.5.3 R-trees 219

4 CURVILINEAR DATA 227
4.1 Strip Trees, Arc Trees, and BSPR 228
4.2 Methods Based on the Region Quadtree 235
4.2.1 Edge Quadtrees 235
4.2.2 Line Quadtrees 237
4.2.3 PM Quadtrees 239
4.2.3.1 The PM_1 Quadtree 240
4.2.3.2 The PM_2 Quadtree 257
4.2.3.3 The PM_3 Quadtree 261
4.2.3.4 PMR Quadtrees 264
4.2.3.5 Fragments 269
4.2.3.6 Maintaining Labels of Regions 275
4.2.4 Empirical Comparisons of the Different Representations 278
4.3 Methods Rooted in Computational Geometry 286
4.3.1 The K-structure 287

4.3.2 Separating Chains and Layered Dags 293
4.3.3 Comparison with PM Quadtrees 306
4.4 Conclusion 312

5 VOLUME DATA 315
5.1 Solid Modeling 316
5.2 Region Octrees 318
5.3 PM Octrees 326
5.4 Boundary Model (BRep) 331
5.5 Constructive Solid Geometry (CSG) 338
5.5.1 CSG Evaluation by Bintree Conversion 340
5.5.1.1 Algorithm for a Single Halfspace 341
5.5.1.2 Algorithm for a CSG Tree 346
5.5.1.3 Incorporation of the Time Dimension 355
5.5.2 PM-CSG Trees 360
5.6 Surface-based Object Representations 365
5.7 Prism Trees 370
5.8 Cone Trees 374

Solutions to Exercises 377
Appendix: Description of Pseudo-Code Language 411
References 415
Name and Credit Index 465
Subject Index 477

1 INTRODUCTION

There are numerous hierarchical data structuring techniques in use for representing spatial data. One commonly used technique is the quadtree, which has evolved from work in different fields. Thus it is natural that a number of adaptations of it exist for each spatial data type. Its development has been motivated to a large extent by a desire to save storage by aggregating data having identical or similar values. We will see, however, that this is not always the case. In fact, the savings in execution time that arise from this aggregation are often of equal or greater importance.

In this chapter we start with a historical overview of quadtrees, including definitions. Since the primary focus in this book is on the representation of regions, what follows is a discussion of region representation in the context of different space decomposition methods. This is done by examining polygonal and nonpolygonal tilings of the plane. The emphasis is on justifying the use of a decomposition into squares. We conclude with a detailed analysis of the space requirements of the quadtree representation.

Most of the presentation in this chapter is in the context of two-dimensional regions. The extension of the topics in this chapter, and remaining chapters, to three-dimensional region data, and higher, is straightforward and, aside from definitions, is often left to the exercises. Nevertheless, the concept of an octree, a quadtree-like representation of three-dimensional regions, is defined and a brief explanation is given of how some of the results described here are applicable to higher-dimensional data.

1.1 BASIC DEFINITIONS

First, we define a few terms with respect to two-dimensional data. Assume the existence of an array of picture elements (termed *pixels*) in two dimensions. We use the term *image* to refer to the original array of pixels. If its elements are black or

white, then it is said to be *binary*. If shades of gray are possible (i.e., gray levels), the image is said to be a *gray–scale* image. In the discussion, we are primarily concerned with binary images. Assume that the image is on an infinite background of white pixels. The *border* of the image is the outer boundary of the square corresponding to the array.

Two pixels are said to be 4-*adjacent* if they are adjacent to each other in the horizontal or vertical direction. If the concept of adjacency also includes adjacency at a corner (i.e., diagonal adjacencies), then the pixels are said to be 8-*adjacent*. A set S is said to be *four–connected* (*eight–connected*) if for any pixels p, q in S there exists a sequence of pixels $p = p_0, p_1, \cdots, p_n = q$ in S, such that p_{i+1} is 4-adjacent (8-adjacent) to p_i, $0 \leq i < n$.

A black *region*, or black four-connected *component*, is a maximal four-connected set of black pixels. The process of assigning the same label to all 4-adjacent black pixels is called *connected component labeling* (see Chapter 5 of [Same90b]). A white *region* is a maximal *eight–connected* set of white pixels defined analogously. The complement of a black region consists of a union of eight-connected white regions. Exactly one of these white regions contains the infinite background of white pixels. All the other white regions, if any, are called *holes* in the black region. The black region, say R, is surrounded by the infinite white region and R surrounds the other white regions, if any.

A pixel is said to have four edges, each of which is of unit length. The *boundary* of a black region consists of the set of edges of its constituent pixels that also serve as edges of white pixels. Similar definitions can be formulated in terms of rectangular blocks, all of whose pixels are identically colored. For example, two disjoint blocks, P and Q, are said to be 4-*adjacent* if there exists a pixel p in P and a pixel q in Q such that p and q are 4-adjacent. Eight-adjacency for blocks (as well as connected component labeling) is defined analogously.

1.2 OVERVIEW OF QUADTREES AND OCTREES

The term *quadtree* is used to describe a class of hierarchical data structures whose common property is that they are based on the principle of recursive decomposition of space. They can be differentiated on the following bases:

1. The type of data they are used to represent
2. The principle guiding the decomposition process
3. The resolution (variable or not)

Currently they are used for point data, areas, curves, surfaces, and volumes. The decomposition may be into equal parts on each level (i.e., regular polygons and termed a *regular decomposition*), or it may be governed by the input. In computer graphics this distinction is often phrased in terms of image-space hierarchies versus object-space hierarchies, respectively [Suth74]. The resolution of the decomposition

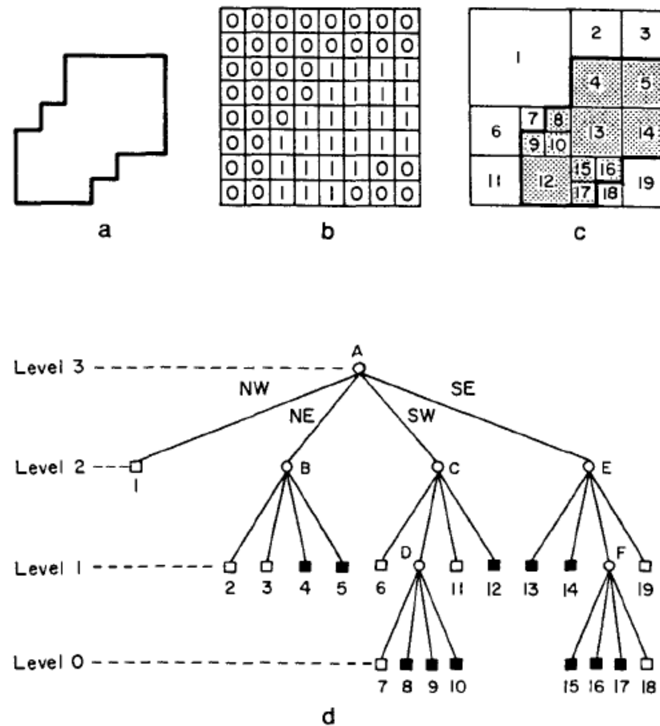


Figure 1.1 An example of (a) a region, (b) its binary array, (c) its maximal blocks (blocks in the region are shaded), and (d) the corresponding quadtree

(i.e., the number of times that the decomposition process is applied) may be fixed beforehand, or it may be governed by properties of the input data. For some applications we can also differentiate the data structures on the basis of whether they specify the boundaries of regions (e.g., curves and surfaces) or organize their interiors (e.g., areas and volumes).

The first example of a quadtree representation of data is concerned with the representation of two-dimensional binary region data. The most studied quadtree approach to region representation, called a *region quadtree* (but often termed a *quadtree* in the rest of this chapter), is based on the successive subdivision of a bounded image array into four equal-sized quadrants. If the array does not consist entirely of 1s or entirely of 0s (i.e., the region does not cover the entire array), then it is subdivided into quadrants, subquadrants, and so on, until blocks are obtained that consist entirely of 1s or entirely of 0s; that is, each block is entirely contained in the region or entirely disjoint from it. The region quadtree can be characterized as a variable resolution data structure.

As an example of the region quadtree, consider the region shown in Figure 1.1a represented by the $2^3 \times 2^3$ binary array in Figure 1.1b. Observe that the 1s correspond to picture elements (i.e., pixels) in the region, and the 0s correspond to picture elements outside the region. The resulting blocks for the array of Figure 1.1b are shown in Figure 1.1c. This process is represented by a tree of degree 4 (i.e., each nonleaf node has four sons).

 Unified Patents Exhibit 1005 App'x A-N

In the tree representation, the root node corresponds to the entire array. Each son of a node represents a quadrant (labeled in order NW, NE, SW, SE) of the region represented by that node. The leaf nodes of the tree correspond to those blocks for which no further subdivision is necessary. A leaf node is said to be black or white depending on whether its corresponding block is entirely inside (it contains only 1s) or entirely outside the represented region (it contains no 1s). All nonleaf nodes are said to be gray (i.e., its block contains 0s and 1s). Given a $2^n \times 2^n$ image, the root node is said to be at level n while a node at level 0 corresponds to a single pixel in the image.[1] The region quadtree representation for Figure 1.1c is shown in Figure 1.1d. The leaf nodes are labeled with numbers, while the nonleaf nodes are labeled with letters. The levels of the tree are also marked.

Our definition of the region quadtree implies that it is constructed by a top-down process. In practice, the process is bottom-up, and one usually uses one of two approaches. The first approach [Same80b] is applicable when the image array is not too large. In such a case, the elements of the array are inspected in the order given by the labels on the array in Figure 1.2 (which corresponds to the image of Figure 1.1a). This order is also known as a Morton order [Mort66] (discussed in Section 1.3). By using such a method, a leaf node is never created until it is known to be maximal. An equivalent statement is that the situation does not arise in which four leaf nodes of the same color necessitate the changing of the color of their parent from gray to black or white as is appropriate. (For more details, see Section 4.1 of [Same90b].)

The second approach [Same81a] is applicable to large images. In this case, the elements of the image are processed one row at a time—for example, in the order given by the labels on the array in Figure 1.3 (which corresponds to the image of Figure 1.1a). This order is also known as a row or raster-scan order (discussed in Section 1.3). A quadtree is built by adding pixel-sized nodes one by one in the order in which they appear in the file. (For more details, see Section 4.2.1 of [Same90b].) This process can be time-consuming due to the many merging and node insertion operations that need to take place.

The above method has been improved by using a predictive method [Shaf86a, Shaf87a], which only makes a single insertion for each node in the final quadtree and performs no merge operations. It is based on processing the image in row order (top to bottom, left to right), always inserting the largest node (i.e., block) for which the current pixel is the first (upper leftmost) pixel. Such a policy avoids the necessity of merging since the upper leftmost pixel of any block is inserted before any other pixel of that block. Therefore it is impossible for four sibling nodes to be of the same color. This method makes use of an auxiliary array of size $O(2^n)$ for a $2^n \times 2^n$ image. (For more details, see Section 4.2.3 of [Same90b].)

The region quadtree is easily extended to represent three-dimensional binary region data and the resulting data structure is called a *region octree* (termed an *octree*

[1] Alternatively we can say that the root node is at depth 0 while a node at depth n corresponds to a single pixel in the image. In this book both concepts of level and depth are used to describe the relative position of nodes. The one that is chosen is context dependent.

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

Figure 1.2 Morton order for the pixels of Figure 1.1

in the rest of this chapter). We start with a $2^n \times 2^n \times 2^n$ object array of unit cubes (termed *voxels* or *obels*). The octree is based on the successive subdivision of an object array into octants. If the array does not consist entirely of 1s or entirely of 0s, it is subdivided into octants, suboctants, and so on until cubes (possibly single voxels) are obtained that consist of 1s or of 0s; that is, they are entirely contained in the region or entirely disjoint from it.

This subdivision process is represented by a tree of degree 8 in which the root node represents the entire object and the leaf nodes correspond to those cubes of the array for which no further subdivision is necessary. Leaf nodes are said to be black or white (alternatively, full or void) depending on whether their corresponding cubes are entirely within or outside the object, respectively. All nonleaf nodes are said to be gray. Figure 1.4a is an example of a simple three-dimensional object, in the form of a staircase, whose octree block decomposition is given in Figure 1.4b and whose tree representation is given in Figure 1.4c.

The region quadtree is a member of a class of representations characterized as being a collection of maximal (according to an appropriate definition) blocks, each of which is contained in a given region and whose union is the entire region. The simplest such representation is the runlength code, where the blocks are restricted to $1 \times m$ rectangles [Ruto68]. A more general representation treats the region as a union of maximal square blocks (or blocks of any other desired shape) that may possibly overlap. Usually the blocks are specified by their centers and radii. This representation is called the *medial axis transformation* (*MAT*) [Blum67, Rose66]. Of course, other approaches are also possible (e.g., rectangular coding [Kim83, Kim86], TID [Scot85, Scot86]).

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

Figure 1.3 Raster-scan order for the pixels of Figure 1.1

Unified Patents Exhibit 1005 App'x A-N

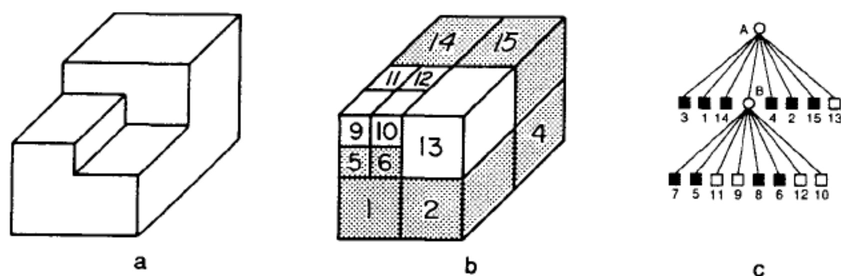


Figure 1.4 (a) Example three-dimensional object; (b) its octree block decomposition; (c) its tree representation

The region quadtree is a variant on the maximal block representation. It requires the blocks to be disjoint and to have standard sizes (i.e., sides of lengths that are powers of two) and standard locations. The motivation for its development is a desire to obtain a systematic way to represent homogeneous parts of an image. Thus to transform the data into a region quadtree, a criterion must be chosen for deciding that an image is homogeneous (i.e., uniform).

One such criterion is that the standard deviation of its gray levels is below a given threshold t. Using this criterion, the image array is successively subdivided into quadrants, subquadrants, and so on until homogeneous blocks are obtained. This process leads to a regular decomposition. If one associates with each leaf node the mean gray level of its block, the resulting region quadtree will then completely specify a piecewise approximation to the image where each homogeneous block is represented by its mean. The case where $t=0$ (i.e., a block is not homogeneous unless its gray level is constant) is of particular interest since it permits an exact reconstruction of the image from its quadtree.

Note that the blocks of the region quadtree do not necessarily correspond to maximal homogeneous regions in the image. Most likely there exist unions of the blocks that are still homogeneous. To obtain a segmentation of the image into maximal homogeneous regions, we must allow merging of adjacent blocks (or unions of blocks) as long as the resulting region remains homogeneous. This is achieved by a 'split-and-merge' algorithm [Horo76]. However, the resulting partition will no longer be represented by a quadtree; instead the final representation is in the form of an adjacency graph. Thus the region quadtree is used as an initial step in the segmentation process.

For example, Figure 1.5b–d demonstrates the results of the application, in sequence, of merging, splitting, and grouping to the initial image decomposition of Figure 1.5a. In this case, the image is initially decomposed into 16 equal-sized square blocks. Next the 'merge' step attempts to form larger blocks by recursively merging groups of four homogeneous 'brothers' (the four blocks in the NW and SE quadrants of Figure 1.5b). The 'split' step recursively decomposes blocks that are not homogeneous (the NE and SW quadrants of Figure 1.5c) until a particular homogeneity criterion is satisfied or a given level is encountered. Finally the 'grouping' step aggregates all homogeneous 4-adjacent black blocks into one region apiece;

 Unified Patents Exhibit 1005 App'x A-N

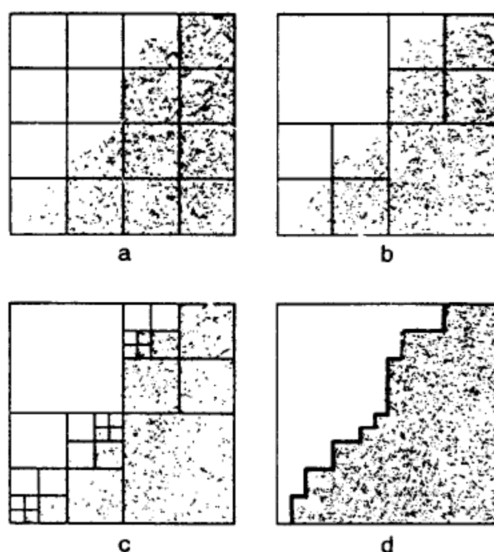


Figure 1.5 Example illustrating the 'split-and-merge' segmentation procedure: (a) start, (b) merge, (c) split, (d) grouping

the 8-adjacent white blocks are similarly aggregated into white regions (Figure 1.5d).

An alternative to the region quadtree representation is to use a decomposition method that is not regular (i.e., rectangles of arbitrary size rather than squares). This alternative has the potential of requiring less space. Its drawback is that the determination of optimal partition points may be computationally expensive (see Exercise 1.10). A closely related problem, decomposing a region into a minimum number of rectangles, is known to be NP-complete[2] [Gare79] if the region is permitted to contain holes [Ling82].

The homogeneity criterion ultimately chosen to guide the subdivision process depends on the type of region data represented. In the remainder of this chapter we shall assume that the domain is a $2^n \times 2^n$ binary image with 1, or black, corresponding to foreground and 0, or white, corresponding to background (e.g., Figure 1.1).

[2] A problem is in NP if it can be solved nondeterministically in polynomial time. A nondeterministic solution process proceeds by 'guessing' a solution and then verifying that the solution is correct. Assume that n is the size of the problem (e.g., for sorting, n is the number of records to be sorted). Intuitively, then, a problem is in NP if there is a polynomial $P(n)$ such that if one guesses a solution, it can be verified in $O(P(n))$ time, whether the guess is indeed a correct solution. Thus the verification process is the key to determining whether a problem is in NP, not the actual solution of the problem.

A problem is NP-complete if it is 'at least as hard' as any other problem in NP. Somewhat more formally, a problem P_1 in NP is NP-complete if the following property holds: for all other problems P_i in NP, if P_1 can be solved deterministically in $O(f(n))$ time, then P_i can be solved in $O(P(f(n)))$ time for some polynomial P. It has been conjectured that no NP-complete problem can be solved deterministically in polynomial time, but this is not known for sure. The theory of NP-completeness is discussed in detail in [Gare79].

Nevertheless the quadtree and octree can be used to represent multicolored data (e.g., a landuse class map associating colors with crops [Same87a]).

It is interesting to note that Kawaguchi, Endo, and Matsunaga [Kawa83] use a sequence of m binary-valued quadtrees to encode image data of 2^m gray levels, where the various gray levels are encoded by use of Gray codes (see, e.g., [McCl65]). This should lead to compaction (i.e., larger-sized blocks) since the Gray code guarantees that the binary representation of the codes of adjacent gray level values differ by only one binary digit.[3] Note, though, that if the primary interest is in image compression, there exist even better methods (see, e.g., [Prat78]); however, they are beyond the scope of this book (but see Chapter 8 of [Same90b]). In another context, Kawaguchi, Endo, and Yokota [Kawa80b] point out that a sequence of related images (e.g., in an animation application) can be stored compactly as a sequence of quadtrees such that the ith element is the result of exclusive oring the first i images (see Exercise 1.7).

Unfortunately the term *quadtree* has taken on more than one meaning. The region quadtree, as described earlier, is a partition of space into a set of squares whose sides are all a power of two long. This formulation is due to Klinger [Klin71] and Klinger and Dyer, who used the term *Q-tree* [Klin76], whereas Hunter [Hunt78] was the first to use the term *quadtree* in such a context. Actually a more precise term would be *quadtrie*, as it is really a trie structure [Fred60] in two dimensions.[4] A similar partition of space into rectangular quadrants, also termed a *quadtree*, was used by Finkel and Bentley [Fink74]. It is an adaptation of the binary search tree [Knut73b] to two dimensions (which can be easily extended to an arbitrary number of dimensions). It is primarily used to represent multidimensional point data, and we shall refer to it as a *point quadtree* where confusion with a region quadtree is possible.

As an example of a point quadtree, consider Figure 1.6, which is built for the sequence Chicago, Mobile, Toronto, Buffalo, Denver, Omaha, Atlanta, and Miami[5]

---

[3] The Gray code is motivated by a desire to reduce errors in transitions between successive gray level values. Its one bit difference guarantee is achieved by the following encoding. Consider the binary representation of the integers from 0 to $2^m - 1$. This representation can be obtained by constructing a binary tree, say T, of height m where each left branch is labeled 0 while each right branch is labeled 1. Each leaf node, say P, is given the label formed by concatenating the labels of the branches taken by the path from the root to P. Enumerating the leaf nodes from left to right yields the binary integers 0 to $2^m - 1$. The Gray codes of the integers are obtained by constructing a new binary tree, say T', such that the labels of some of the branches in T' are the reverse of what they were in T. The algorithm is as follows. Initially, T' is a copy of T. Next, traverse T in preorder (i.e., visit the root node, followed by the left and right subtrees). For each branch in T labeled 1, exchange the labels of the two descendant branches of its corresponding branch in T'. No action is taken for descendants of branches in T labeled 0. Enumerating the leaf nodes in T' from left to right yields the Gray codes of the integers 0 to $2^m - 1$. For example, for 8 gray levels (i.e., $m = 3$), we have 000, 001, 011, 010, 110, 111, 101, 100.

[4] In a one-dimensional *trie* structure, each data item or key is treated as a sequence of characters where each character has M possible values. A node at depth i in the trie represents an M-way branch depending on the ith character. The data are stored in the leaf nodes, and the shape of the trie is independent of the order in which the data are processed. Such a structure is also known as a *digital tree* [Knut73b].

[5] The correspondence between coordinate values and city names is not geographically correct. This liberty has been taken so that the same example can be used throughout the text to illustrate a variety of concepts.

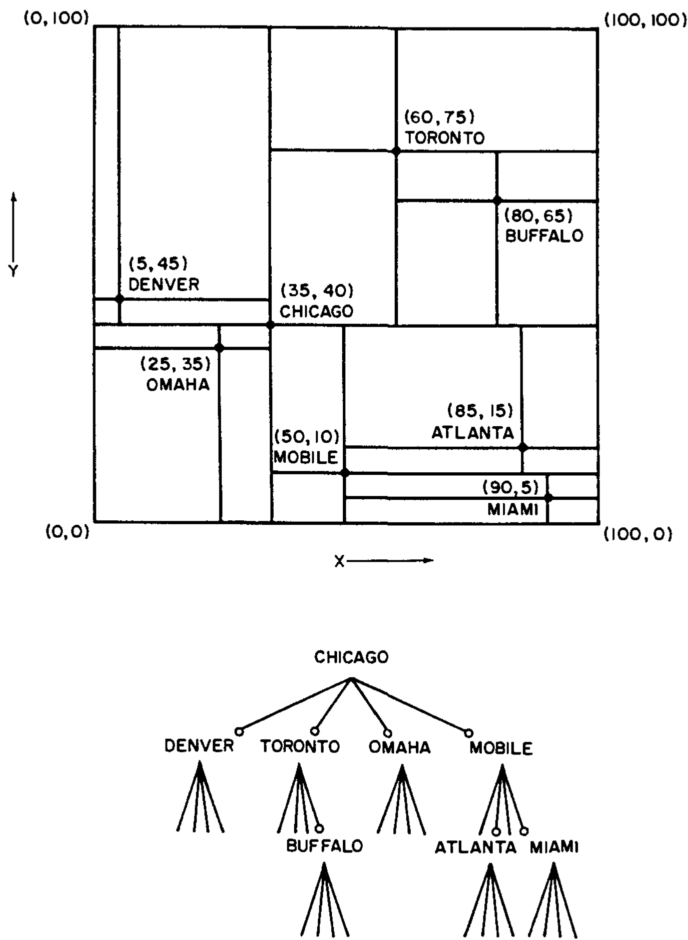


Figure 1.6 A point quadtree and the records it represents

in the order in which they are listed here.[6] Its shape is highly dependent on the order in which the points are added to it. Of course, trie-based point representations also exist (see Sections 2.6.1 and 2.6.2).

Exercises

1.1. The region quadtree is an alternative to an image representation that is based on the use of an array or even a list. Each of these image representations may be biased in favor of the computation of a particular adjacency relation. Discuss these biases for the array, list, and quadtree representations.

1.2. Given the array representation of a binary image, write an algorithm to construct the corresponding region quadtree.

[6] Refer to Figure 2.5 to see how the point quadtree is constructed in an incremental fashion for Chicago, Mobile, Toronto, and Buffalo.

1.3. Given an image represented by a region quadtree with B black and W white nodes, how many additional nodes are necessary for the nonleaf nodes?
1.4. Given an image represented by a region octree with B black and W white nodes, how many additional nodes are necessary for the nonleaf nodes?
1.5. Suppose that an octree is used to represent a collection of disjoint spheres. What would you use as a leaf criterion?
1.6. The quadtree can be generalized to represent data in arbitrary dimensions. As we saw, the octree is its three-dimensional analog. The renowned artist Escher [Coxe86] is noted for etchings of unusual interpretations of geometric objects such as staircases. How would you represent one of Escher's staircases?
1.7. Let $\oplus$ denote an exclusive or operation. Given a sequence of related images, $<P_n, P_{n-1}, \cdots, P_0>$, define another sequence $<Q_n, Q_{n-1}, \cdots, Q_0>$ such that $Q_0 = P_0$ and $Q_i = P_i \oplus Q_{i-1}$ for $i > 0$. Show that when the sequences P and Q are represented as quadtrees, replacing sequence P by sequence Q results in fewer nodes.
1.8. Prove that in Exercise 1.7 the sequence P can be reconstructed from the sequence Q. In particular, given Q_i and Q_{i-1}, determine P_i.
1.9. Write an algorithm to construct the Gray codes of the integers 0 to 2^m-1.
1.10. Find a polynomial-time algorithm to decompose a region optimally so that its quadtree representation uses a minimum amount of space (i.e., a minimum number of nodes). In this case, you can assume that the decomposition lines can be placed in arbitrary positions so that the space requirement is reduced. In other words, the decomposition lines need not split the space into four squares of equal size. Thus the decomposition is similar to that induced by a point quadtree.

1.3 HISTORY OF THE USE OF QUADTREES AND OCTREES

The origin of the principle of recursive decomposition, upon which all quadtrees are based, is difficult to ascertain. Below, to give some indication of the uses of the region quadtree, some of its applications to geometric data are traced briefly. Most likely it was first seen as a way of aggregating blocks of zeros in sparse matrices. Indeed Hoare [Hoar72] attributes a one-level decomposition of a matrix into square blocks to Dijkstra. Morton [Mort66] used it as a means of indexing into a geographic database (i.e., it acts as a spatial index).

Warnock, in a pair of reports that serve as landmarks in computer graphics [Warn68, Warn69b], described the implementation of hidden-line and hidden-surface elimination algorithms using a recursive decomposition of the picture area. The picture area is repeatedly subdivided into rectangles that are successively smaller while searching for areas that are sufficiently simple to be displayed. Klinger [Klin71] and Klinger and Dyer [Klin76] applied these ideas to pattern recognition and image processing, while Hunter [Hunt78] used them for an animation application.

The SRI robot project [Nils69] used a three-level decomposition of space to represent a map of the robot's world. Eastman [East70] observes that recursive decomposition might be used for space planning in an architectural context and presents a simplified version of the SRI robot representation. A quadtree-like representation in the form of production rules called DF-expressions (denoting 'depth-first') is discussed by Kawaguchi and Endo [Kawa80a] and Kawaguchi, Endo, and Yokota

 Unified Patents Exhibit 1005 App'x A-N

[Kawa80b] (see also Section 1.5). Tucker [Tuck84a] uses quadtree refinement as a control strategy for an expert vision system.

The three-dimensional variant of the region quadtree—the octree—was developed independently by a number of researchers. Hunter [Hunt78] mentioned it as a natural extension of the quadtree. Reddy and Rubin [Redd78] proposed the octree as one of three representations for solid objects. The second is a three-dimensional generalization of the point quadtree of Finkel and Bentley [Fink74]—that is, a decomposition into rectangular parallelepipeds (as opposed to cubes) with planes perpendicular to the x, y, and z axes. The third breaks the object into rectangular parallelepipeds that are not necessarily aligned with an axis. The parallelepipeds are of arbitrary sizes and orientations. Each parallelepiped is recursively subdivided into parallelepipeds in the coordinate space of the enclosing parallelepiped. Reddy and Rubin prefer the third approach for its ease of display.

Situated somewhere between the second and third approaches of Reddy and Rubin is the method of Brooks and Lozano-Perez [Broo83] (see also [Loza81]), who use a recursive decomposition of space into an arbitrary number of rectangular parallelepipeds, with planes perpendicular to the x, y, and z axes, to model space in solving the *findpath* or *piano movers* problem [Schw88] in robotics. This problem arises when planning the motion of a robot in an environment containing known obstacles and the desired solution is a collision-free path obtained by use of a search. Faverjon [Fave84] discusses an approach to this problem that uses an octree, as do Samet and Tamminen [Same85g] and Fujimura and Samet [Fuji89].

Jackins and Tanimoto [Jack80] adapted Hunter and Steiglitz's quadtree translation algorithm [Hunt78, Hunt79b] to objects represented by octrees. Meagher [Meag82a] developed numerous algorithms for performing solid modeling operations in an environment where the octree is the underlying representation. Yau and Srihari [Yau83] extended the octree to arbitrary dimensions in the process of developing algorithms to handle medical images.

Both quadtrees and octrees are frequently used in the construction of meshes for finite element analysis. The use of recursive decomposition for meshes was initially suggested by Rheinboldt and Mesztenyi [Rhei80]. Yerry and Shephard [Yerr83] adapted the quadtree and octree to generate meshes automatically for three-dimensional solids represented by a superquadric surface-based modeler. This has been extended by Kela, Voelcker, and Goldak [Kela84b] (see also [Kela86]) to mesh boundary regions directly, rather than through discrete approximations, and to facilitate incremental adaptive analysis by exploiting the spatial index nature of the quadtree and octree.

Parallel to the development of the quadtree and octree data structures, there has been related work by researchers in the field of image understanding. Kelly [Kell71] introduced the concept of a *plan*, which is a small picture whose pixels represent gray-scale averages over 8×8 blocks of a larger picture. Needless effort in edge detection is avoided by first determining edges in the plan and then using these edges to search selectively for edges in the larger picture. Generalizations of this idea motivated the development of multiresolution image representations—for example,

 Unified Patents Exhibit 1005 App'x A-N

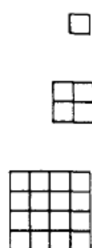


Figure 1.7 Structure of a pyramid having three levels

the recognition cone of Uhr [Uhr72], the preprocessing cone of Riseman and Arbib [Rise77], and the pyramid of Tanimoto and Pavlidis [Tani75]. Of these representations, the pyramid is the closest relative of the region quadtree.

Given a $2^n \times 2^n$ image array, say $A(n)$, a *pyramid* is a sequence of arrays $\{A(i)\}$ such that $A(i-1)$ is a version of $A(i)$ at half the scale of $A(i)$. $A(0)$ is a single pixel. Figure 1.7 shows the structure of a pyramid having three levels. It should be clear that a pyramid can also be defined in a more general way by permitting finer scales of resolution than the power of two scale.

At times, it is more convenient to define a pyramid in the form of a tree. Again, assuming a $2^n \times 2^n$ image, a recursive decomposition into quadrants is performed, just as in quadtree construction, except that we keep subdividing until we reach the individual pixels. The leaf nodes of the resulting tree represent the pixels, while the nodes immediately above the leaf nodes correspond to the array $A(n-1)$, which is of size $2^{n-1} \times 2^{n-1}$. The nonleaf nodes are assigned a value that is a function of the nodes below them (i.e., their sons) such as the average gray level. Thus we see that a pyramid is a multiresolution representation, whereas the region quadtree is a variable

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

Figure 1.8 Example pyramid A(3)

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

Figure 1.9 A(2) corresponding to Figure 1.8

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

Figure 1.10 The overlapping blocks in which pixel 28 participates

resolution representation. Another analogy is that the pyramid is a complete quadtree [Knut73a].

The above definition of a pyramid is based on nonoverlapping 2×2 blocks of pixels. An alternative definition, termed an *overlapping pyramid*, uses overlapping blocks of pixels. One of the simplest schemes makes use of 4×4 blocks that overlap by 50% in both the horizontal and vertical directions [Burt81]. For example, Figure 1.8 is a $2^3 \times 2^3$ array, say A(3), whose pixels are labeled 1-64. Figure 1.9 is A(2) corresponding to Figure 1.8 with elements labeled A-P. The 4×4 neighborhood corresponding to element F in Figure 1.9 consists of pixels 10–13, 18–21, 26–29, and 34–37. This method implies that each block at a given level participates in four blocks at the immediately higher level. Thus the containment relations between blocks no longer form a tree. For example, pixel 28 participates in blocks F, G, J, and K in the next higher level (see Figure 1.10 where the four neighborhoods corresponding to F, G, J, and K are drawn as squares).

To avoid treating border cases differently, each level in the overlapped pyramid is assumed to be cyclically closed (i.e., the top row at each level is adjacent to the bottom row and similarly for the columns at the extreme left and right of each level). Once again we say that the value of a node is the average of the values of the nodes in its block on the immediately lower level. The overlapped pyramid may be compared with the Quadtree Medial Axis Transform (see Section 9.3.1 of [Same90b]) in the sense that both may result in nondisjoint decompositions of space.

Pyramids have been applied to the problems of feature detection and extraction since they can be used to limit the scope of the search. Once a piece of information of interest is found at a coarse level, the finer resolution levels can be searched. This approach was followed by Davis and Roussopoulos [Davi80] in approximate pattern matching. Pyramids can also be used for encoding information about edges, lines, and curves in an image [Shne81c, Krop86]. One note of caution: the reduction of resolution has an effect on the visual appearance of edges and small objects [Tani76]. In particular, at a coarser level of resolution, edges tend to get smeared, and region separation may disappear. Pyramids have also been used as the starting point for a 'split-and-merge' segmentation algorithm [Piet82].

Quadtree-like decompositions are useful as space-ordering methods. The purpose is to optimize the storage and processing sequences for two-dimensional data by mapping them into one dimension (i.e., linearizing them). This mapping should pre-

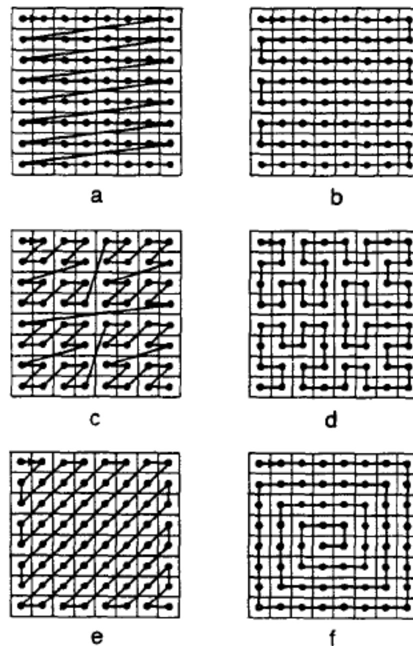


Figure 1.11 The result of applying a number of different space-ordering methods to an 8 × 8 image whose first element is in the upper left corner of the image: (a) row order, (b) row-prime order, (c) Morton order, (d) Peano-Hilbert order, (e) Cantor-diagonal order, (f) spiral order

serve the spatial locality of the original two-dimensional image in one dimension. The result of the mapping is also known as a *space-filling curve* [Gold81, Witt83] because it passes through every point in the image.

Goodchild and Grandfield [Good83] discuss a number of space-ordering methods, some of which are illustrated in Figure 1.11. Each has different characteristics. The row (Figure 1.11a), also known as raster-scan, and row-prime orders (Figure 1.11b) are similar in the same way as are the Morton [Mort66, Pean90] (Figure 1.11c) and the Peano-Hilbert [Hilb91] (Figure 1.11d) orders. The primary difference is that in both the row-prime and Peano-Hilbert orders every element is a 4-adjacent neighbor of the previous element in the sequence, and thus they have a slightly higher degree of locality than the row and Morton orders, respectively. Both the Morton and Peano-Hilbert orders exhaust a quadrant or subquadrant of a square image before exiting it. They are both related to quadtrees; however, as we saw above, the Morton order does not traverse the image in a spatially contiguous manner (the result has the shape of the letter 'N' or 'Z' and is also known as N order [Whit82] and Z order [Oren84]).

For both the Morton and Peano-Hilbert orders, there is no need to know the maximum values of the coordinates. The Morton order is symmetric, while the Peano-Hilbert order is not. One advantage of the Morton order is that the position of each element in the ordering (termed its *key*) can be determined by interleaving the

bits of the x and y coordinates of the element; this is not easy for the Peano-Hilbert order. Another advantage of the Morton order is that the recursion necessary for its generation is quite easy to specify.

Other orders are the Cantor-diagonal order (Figure 1.11e) and the spiral order (Figure 1.11f). The Cantor-diagonal order proceeds outward from the origin and visits the elements in an order similar to row-prime with the difference that elements are visited in order of their increasing 'Manhattan' (or 'city block') distance.[7] Thus it is good for ordering a space that is unbounded in the two directions emanating from the origin which has been relocated to the center of the image. On the other hand, the spiral order is attractive when ordering a space that is unbounded in the four directions emanating from the origin.

The most interesting orders, as far as we are concerned, are the Morton and Peano-Hilbert orders since they can also be used to order a space that has been aggregated into squares. Of these two orderings, the Morton order is by far the more frequently used as a result of the simplicity of the conversion process between the key and its corresponding element in the multidimensional space. In this book we are primarily interested in Morton orderings. (For further discussion of some of the properties of these two orderings, see [Patr68, Butz71, Alex79, Alex80, Laur85].)

Exercises

1.11. Write an algorithm to extract the x and y coordinates from a Peano-Hilbert order key.

1.12. Write an algorithm to construct the Peano-Hilbert key for a given point (x, y). Try to make it optimal.

1.13. Suppose that you are given a $2^n \times 2^n$ array of points such that the horizontal and vertical distances between 4-adjacent points are 1. What is the average distance between successive points when the points are ordered according to the orders illustrated in Figure 1.11? What about a random order?

1.14. Suppose that you are given a $2^n \times 2^n$ image. Assume that the image is stored on disk in pages of size $2^m \times 2^m$ where n is much larger than m. What is the average cost of retrieving a pixel and its 4-adjacent neighbors when the image is ordered according to the orders illustrated in Figure 1.11?

1.15. The traveling salesman problem [Lawl85] is one where a set of points is given and it is desired to find the path of minimum distance such that each point is visited only once. This is an NP-complete problem [Gare79] and thus there is a considerable amount of work in formulating approximate solutions to it [Bent82]. For example, consider the following approximate solution. Assume that the points are uniformly distributed in the unit square. Let d be the expected Euclidean distance between two independent points. Now, sort the points using the row order and the Morton order. Laurini [Laur85] simulated the average Euclidean distance between successive points in these orders and found it to be $d/2$ for the row order and $d/3$ for the Morton order. Can you derive these averages analytically? What are the average values for the other orders illustrated in Figure 1.11? What about a random order?

---

[7] The Manhattan distance between points (x_1, y_1) and (x_2, y_2) is $|x_1 - x_2| + |y_1 - y_2|$ (for more details, see Section 9.1 of [Same90b]).

1.16. Suppose that the traveling salesman problem is solved using a traversal of the points in Morton order as discussed in Exercise 1.15. In particular, assume that the set of points is decomposed in such a way that each square block contains just one point. This yields a point representation that is analogous to the region quadtree (termed a PR quadtree and discussed in Section 2.6.2). How close does such a solution come to optimality?

1.4 SPACE DECOMPOSITION METHODS

In general, any planar decomposition used as a basis for an image representation should possess the following two properties:

1. The partition should be an infinitely repetitive pattern so that it can be used for images of any size.
2. The partition should be infinitely decomposable into increasingly finer patterns (i.e., higher resolution).

In this section, the discussion is restricted to two-dimensional data. Thus we are dealing with planar space decompositions. Space decompositions can be classified into two categories, depending on the nature of the pattern. The pattern can consist of polygonal shapes or nonpolygonal shapes. The polygonal shapes are generally computationally simpler since their sides can be expressed in terms of linear relations (e.g., equations of lines). They are good for approximating the interior of a region. The nonpolygonal shapes are more flexible since they provide good approximations, in terms of measures, of the boundaries (e.g., perimeter) of regions as well as their interiors (e.g., area).[8]

Moreover, the normals to the boundaries of nonpolygonal shapes are not restricted to a fixed set of directions. For example, in the case of rectangular tiles, there is a 90 degree discontinuity between the normals to boundaries of adjacent tiles. This lack of continuity is a drawback in applications in fields such as computer graphics where such tasks as shading make use of the directions of the surface. However, working with nonpolygonal shapes generally requires use of floating point arithmetic, and hence it is usually more complex.

The remainder of this section expands on a number of polygonal decompositions and compares them. It also contains a brief discussion of one nonpolygonal decomposition that consists of a collection of sector-like objects whose arcs are not necessarily part of a circle. This method is based on polar coordinates where the arc joining two distinct points is formed by linear interpolation. The term *sector tree* is used to describe it. This discussion is of an advanced nature and can be skipped on an initial reading.

[8] Recall the statement in Section 1.2 that hierarchical data structures are often differentiated on the basis of whether they specify the boundaries of regions or organize their interiors.

1.4.1 Polygonal Tilings

Bell, Diaz, Holroyd, and Jackson [Bell83] discuss a number of polygonal tilings of the plane (i.e., tessellations) that satisfy property 1. Figure 1.12 illustrates some of these tessellations. They also present a taxonomy of criteria to distinguish between the various tilings. The tilings, consisting of polygonal tiles, are described by use of a notation based on the degree of each vertex as the edges (i.e., sides) of the 'atomic' tile are visited in order, forming a cycle. For example, the tiling described by $[4.8^2]$ (Figure 1.12c) has the shape of a triangle where the first vertex has degree four while the remaining two vertices have degree eight apiece.

A tiling is said to be *regular* if the atomic tiles are composed of regular polygons (i.e., all sides are of equal length as are the interior angles). A *molecular tile* is an aggregation of atomic tiles to form a hierarchy. It is not necessarily constrained to have the same shape as the atomic tile. When a tile at level k (for all $k > 0$) has the same shape as a tile at level 0 (i.e., it is a scaled image of a tile at level 0), then the tiling is said to be *similar*.

Bell et al. focus on the isohedral tilings where a tiling is said to be *isohedral* if all the tiles are equivalent under the symmetry group of the tiling. A more intuitive

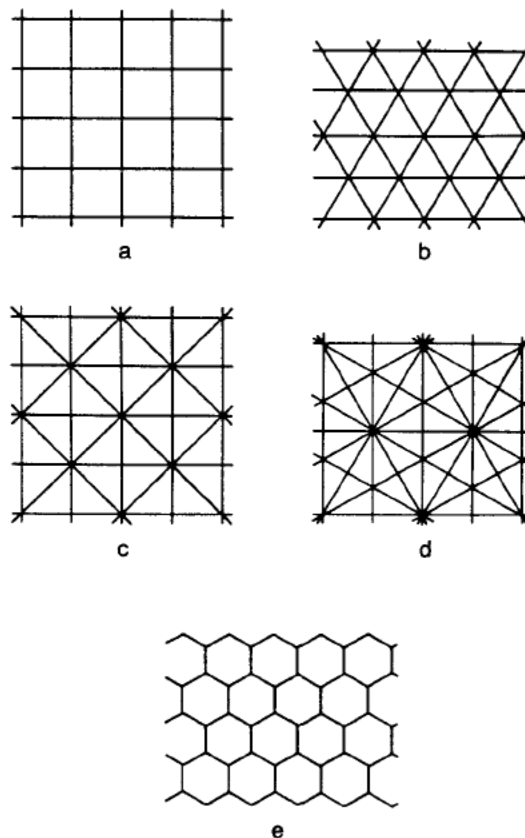


Figure 1.12 Sample tessellations: (a) $[4^4]$ square; (b) $[6^3]$ equilateral triangle; (c) $[4.8^2]$ isoceles triangle; (d) [4.6.12] 30–60 right triangle; (e) $[3^6]$ hexagon

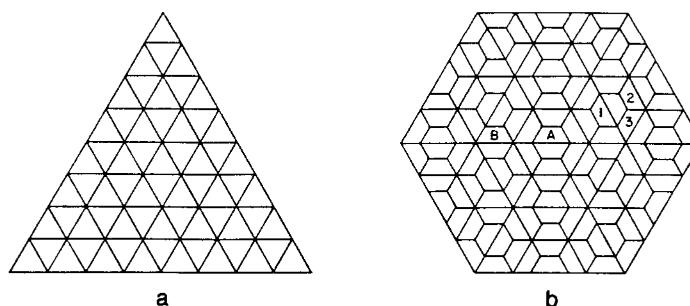


Figure 1.13 Examples of (a) isohedral and (b) nonisohedral tilings

way to conceptualize this definition is to assume the position of an observer who stands in the center of a tile having a given orientation and scans the surroundings. If the view is independent of the tile, the tiling is isohedral. For example, consider the two tilings in Figure 1.13 consisting of triangles (Figure 1.13a) and trapezoids (Figure 1.13b). The triangles are isohedral, whereas the trapezoids are not, as can be seen by the view from tiles A and B.

In the case of the trapezoidal tiling, the viewer from A is surrounded by an infinite number of concentric hexagons, whereas this is not the case for B. In other words, the trapezoidal tiling is not periodic. Also note that all of the tiles in Figure 1.13a are described by $[6^3]$, while those in Figure 1.13b are either $[3^2.4^2]$, $[3^2.6^2]$, or $[3.4.6^2]$ (i.e., tiles labeled 1, 2, and 3, respectively, in Figure 1.13b). When the isohedral tilings are classified by the action of their symmetry group, there are 81 different types [Grün77, Grün87]. When they are classified by their adjacency structure, as done here, there are 11 types.

The most relevant criterion to the discussion is the distinction between limited and unlimited hierarchies of tilings. A *limited* tiling is not similar. A tiling that satisfies property 2 is said to be *unlimited.* Equivalently, in a limited tiling, no change of scale lower than the limit tiling can be made without great difficulty. An alternative characterization of an unlimited tiling is that each edge of a tile lies on an infinite straight line composed entirely of edges. Interestingly the hexagonal tiling $[3^6]$ is limited. Bell et al. claim that only four tilings are unlimited. These are the tilings given in Figure 1.12a–d. Of these, $[4^4]$, consisting of square atomic tiles (Figure 1.12a), and $[6^3]$, consisting of equilateral triangle atomic tiles (Figure 1.12b), are well-known regular tessellations [Ahuj83]. For these two tilings we consider only the molecular tiles given in Figures 1.14a and 1.14b.

The tilings $[4^4]$ and $[6^3]$ can generate an infinite number of different molecular tiles where each molecular tile at the first level consists of n^2 atomic tiles ($n > 1$). The remaining nonregular unlimited triangular tilings, $[4.8^2]$ (Figure 1.12c) and [4.6.12] (Figure 1.12d), are less well understood. One way of generating $[4.8^2]$ and [4.6.12] is to join the centroids of the tiles of $[4^4]$ and $[6^3]$, respectively, to both their vertices and midpoints of their edges. Each of the tilings $[4.8^2]$ and [4.6.12] has two

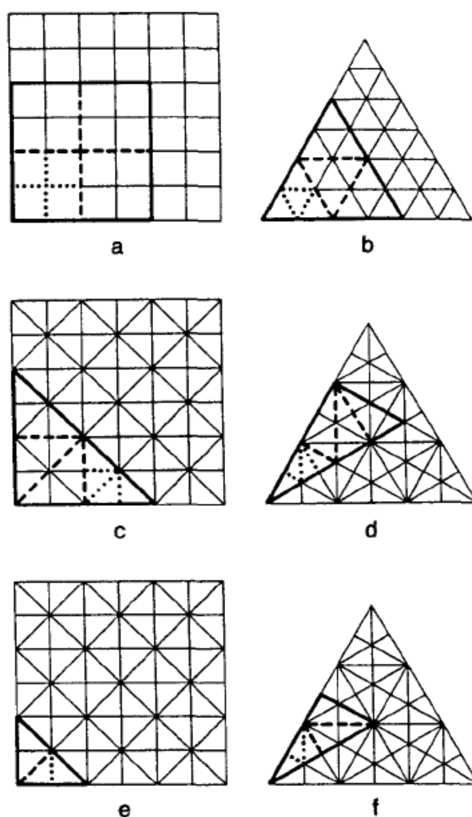


Figure 1.14 Examples illustrating unlimited tilings: (a) [4^4] hierarchy, (b) [6^3] hierarchy, (c) ordinary [4.8^2] hierarchy, (d) ordinary [4.6.12] hierarchy, (e) rotation [4.8^2] hierarchy, (f) reflection [4.6.12] hierarchy

types of hierarchy. [4.8^2] has an ordinary (Figure 1.14c) and a rotation hierarchy (Figure 1.14e) requiring a rotation of 135 degrees between levels. [4.6.12] has an ordinary (Figure 1.14d) and a reflection hierarchy (Figure 1.14f), which requires a reflection of the basic tile between levels.

The distinction between the two types of hierarchies for [4.8^2] and [4.6.12] is necessary because the tiling is not similar without a rotation or a reflection when the hierarchy is not ordinary. This can be seen by observing the use of dots in Figure 1.14 to delimit the atomic tiles in the first molecular tile. Similarly broken lines are used to delimit the components of tiles at the second level (assuming atomic tiles are at level 0). For the ordinary [4.8^2] and [4.6.12] hierarchies, each molecular tile at the first level consists of n^2 ($n > 1$) atomic tiles. In the reflection hierarchy of [4.6.12], each molecular tile at the first level consists of $3 \cdot n^2$ ($n > 1$) atomic tiles, while for the rotation hierarchy of [4.8^2], $2 \cdot n^2$ ($n > 1$) atomic tiles comprise a molecular tile at the first level.

To represent data in the Euclidean plane, any of the unlimited tilings could have been chosen. For a regular decomposition, the tilings [4.8^2] and [4.6.12] are ruled out. Comparing 'square' [4^4] and 'triangular' [6^3] quadtrees, we find that they differ in terms of adjacency and orientation. Let us say that two tiles are *neighbors* if they are

 Unified Patents Exhibit 1005 App'x A-N

adjacent either along an edge or at a vertex. A tiling is *uniformly adjacent* if the distances between the centroid of one tile and the centroids of all its neighbors are the same. The adjacency number of a tiling is the number of different intercentroid distances between any one tile and its neighbors. In the case of [4^4], there are only two adjacency distances, whereas for [6^3] there are three adjacency distances.

A tiling is said to have *uniform orientation* if all tiles with the same orientation can be mapped into each other by translations of the plane that do not involve rotation or reflection. Tiling [4^4] displays uniform orientation, while [6^3] does not. Under the assumption that uniform orientation and a minimal adjacency distance is preferable, we say that [4^4] is more useful than [6^3]. It is also very easy to implement. Nevertheless, [6^3] has its uses. For example, Yamaguchi, Kunii, Fujimura, and Toriya [Yama84] use a triangular quadtree to generate an isometric view from an octree representation of an object (see Section 7.1.4 of [Same90b]).

Of the *limited* tilings, many types of hierarchies may be generated [Bell83]; however, in general, they cannot be decomposed beyond the atomic tiling without changing the basic tile shape. This is a serious deficiency of the hexagonal tessellation [3^6] (Figure 1.12e) since the atomic hexagon can be decomposed only into triangles. Nevertheless the hexagonal tessellation is of considerable interest. It is regular, has a uniform orientation, and, most important, displays a uniform adjacency (i.e., each neighbor of a tile is at the same distance from it).

There are a number of different hexagonal hierarchies distinguished by classifying the shape of the first-level molecular tile on the basis of the number of hexagons that it contains. Three of these tiling hierarchies are given in Figure 1.15 and are called *n-shapes* where *n* denotes the number of atomic tiles in the first-level molecular tile. Of course, these n-shapes are not unique.

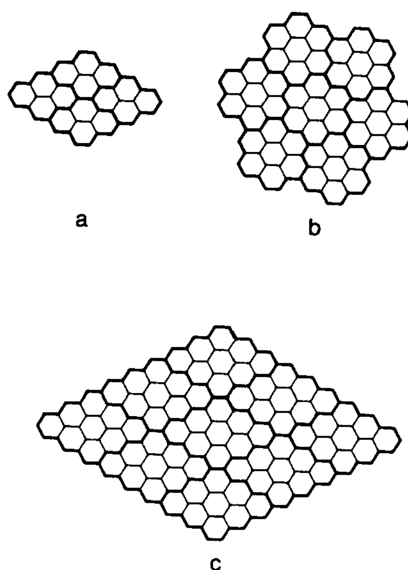


Figure 1.15 Three different hexagonal tiling hierarchies: (a) 4-shape, (b) 7-shape, (c) 9-shape

The 4-shape and the 9-shape have an unusual adjacency property in the sense that no matter how large the molecular tile becomes, contact with two of the tiles (i.e., the one above and the one below) is along only one edge of a hexagonal atomic tile, while contact with the remaining four molecular tiles is along nearly one-quarter of the perimeter of the corresponding molecular tile. The hexagonal pattern of the 4-shape and 9-shape molecular tiles has the shape of a rhombus. In contrast, a 7-shape molecular tile has a uniform contact with its six neighboring molecular tiles.

The type of quadtree used often depends on the grid formed by the image sampling process. Square quadtrees are appropriate for square grids and triangular quadtrees for triangular grids. In the case of a hexagonal grid [Burt80], the 7-shape hierarchy is frequently used since the shape of its molecular tile is more like a hexagon. It is usually described as *rosette*–like (i.e., a *septree*). Note that septrees have jagged edges as they are merged to form larger units (e.g., Figure 1.15b). The septree is used by Gibson and Lucas [Gibs82] (who call it a *generalized balanced ternary* or GBT for short) in the development of algorithms analogous to those existing for quadtrees.

Although the septree can be built up to yield large septrees, the smallest resolution in the septree must be decided upon in advance since its primitive components (i.e., hexagons) cannot later be decomposed into septrees. Therefore the septree yields only a partial hierarchical decomposition in the sense that the components can always be merged into larger units, but they cannot always be broken down. For region data, a pixel is generally an indivisible unit, and thus unlimited decomposition is not absolutely necessary. However, in the case of other data types such as points (see Chapter 2) and lines (see Chapter 4), we will see that the decomposition rules of some representations require that two entities be separated, which may lead to a level of decomposition not known in advance (e.g., a decomposition rule that restricts each square to contain at most one point). In this book the discussion is limited to square quadtrees and their variants.

When the data are spherical, a number of researchers have proposed the use of a representation based on an icosahedron (a 20-faced polyhedron whose faces are regular triangles) [Dutt84, Feke84]. The icosahedron is attractive because, in terms of the number of faces, it is the largest possible regular polyhedron. Each of the triangular faces can be further decomposed in a recursive manner into n^2 ($n > 1$) spherical triangles (the $[6^3]$ tiling).

Fekete and Davis [Feke84] let $n = 2$, which means that at each level of decomposition, three new vertices are generated by halving each side of the triangle; connecting them together yields four triangles. They use the term *property sphere* to describe their representation. The property sphere has been used in object recognition; it is also of potential use in mapping the globe because it can enable accurate modeling of regions around the poles. For example, see Figure 1.16, which is a property sphere representation of some spherical data. In contrast, planar quadtrees are less attractive the farther we get from the equator due to distortions in planarity caused by the earth's curvature. Of course, for true applicability for mapping, we need a closer approximation to a sphere than is provided by the 20 triangles of the icosahedron. Moreover, we want a way to distinguish between different elevations.

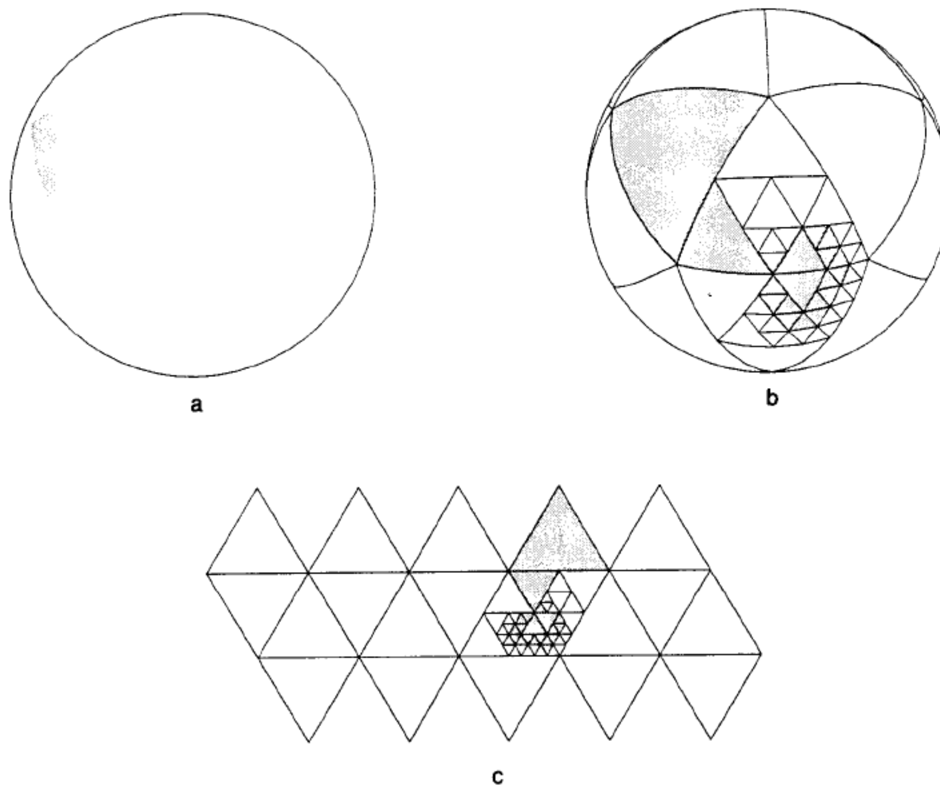


Figure 1.16 Property sphere representation of some spherical data: (a) data, (b) decomposition on a sphere, (c) decomposition on a plane

Dutton [Dutt84] lets $n = \sqrt{3}$, which means that at each level of decomposition, one new vertex is created by connecting the centroid of the triangle to its vertices. The result is an alternating sequence of triangles so that each level is fully contained in the level that was created two steps previously and has nine times as many triangles as that level. Dutton uses the term *triacon* to describe the resulting hierarchy. As an example, consider Figure 1.17, which illustrates four levels of a triacon decomposition. The initial and odd-numbered decompositions are shown with heavy lines, and the even-numbered decompositions are shown with broken and thin lines.

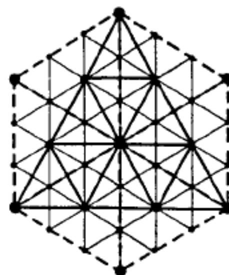


Figure 1.17 Example of a triacon hierarchy

 Unified Patents Exhibit 1005 App'x A-N

The icosahedron is not the only regular polyhedron that can be used to model spherical data. Others include the tetrahedron, hexahedron, octahedron, and dodecahedron, which have 4, 6, 8, and 12 faces, respectively. Collectively these five polyhedra are known as the *Platonic solids* [Peuq84]. The faces of the tetrahedron and octahedron are equilateral triangles, while the faces of the hexahedron and dodecahedron are squares and regular pentagons, respectively.

The dodecahedron is not an appropriate primitive because the pentagonal faces cannot be further decomposed into pentagons or other similar shapes. The tetrahedron and hexahedron (the basis of the octree) have internal angles that are too small to model a sphere properly, thereby leading to shape distortions.

Dutton [Dutt84] points out that the octahedron is attractive for modeling spherical data such as the globe because it can be aligned so that the poles are at opposite vertices and the prime meridian and the equator intersect at another vertex. In addition, one subdivision line of each face is parallel to the equator. Of course, for all of the Platonic solids, only the vertices of the solids touch the sphere; the facets of the solids are interior to the sphere.

Other decompositions for spherical data are also possible. Tobler and Chen [Tobl86] point out the desirability of a close relationship to the commonly used system of latitude and longitude coordinates. In particular, any decomposition that is chosen should enable the use of meridians and parallels to refer to the data. An additional important goal is for the partition to be into units of equal area, which rules out the use of equally spaced lines of latitude (of course, the lines of longitude are equally spaced). In this case, the sphere is projected into a plane using Lambert's cylindrical projection [Adam49], which is locally area preserving. Authalic coordinates [Adam49], which partition the projection into rectangles of equal area, are then derived. (For more details, see [Tobl86].)

The quadtree decomposition has the property that at each subdivision stage, the image is subdivided into four equal-sized parts. When the original image is a square, the result is a collection of squares, each of which has a side whose length is a power of 2. The binary image tree (termed *bintree*) [Know80, Tamm84a, Same88b] is an alternative decomposition defined in a manner analogous to the region quadtree except that at each subdivision stage we subdivide the image into two equal-sized parts. In two dimensions, at odd stages, we partition along the x coordinate, and at even stages, along the y coordinate. The bintree is equivalent to the region quadtree if we replace all leaf nodes at odd stages of subdivision by two identically colored sons.

The bintree is related to the region quadtree in the same way as the k-d tree [Bent75b] (see Section 2.4) is related to the point quadtree [Fink74]. The difference is that region quadtrees and bintrees are used to represent region data with fixed subdivision points, while point quadtrees and k-d trees are used to represent point data where the values of the points determine the subdivision. For example, Figure 1.18 is the bintree representation corresponding to the image of Figure 1.1. We assume that for the x (y) partition, the left subtree corresponds to the west (south) half of the image and the right subtree corresponds to the east (north) half. Once again, as in Figure 1.1, all leaf nodes are labeled with numbers, and the nonleaf nodes are labeled with letters.

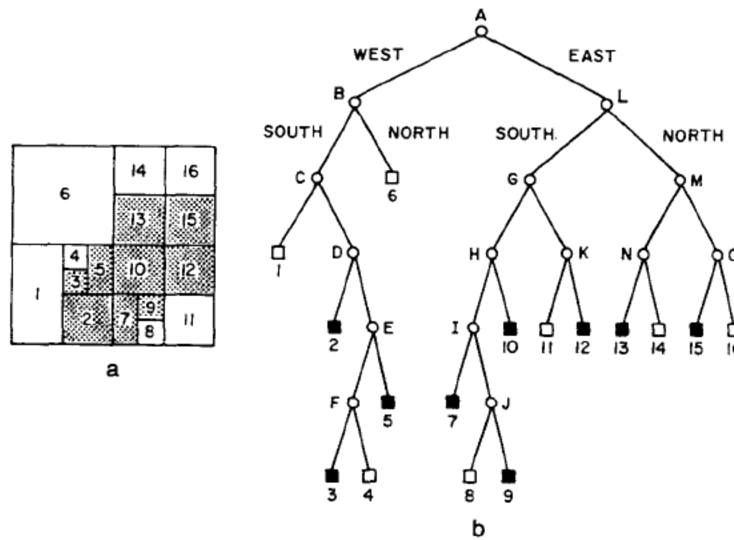


Figure 1.18 Bintree representation corresponding to Figure 1.1: (a) block decomposition, (b) bintree representation of blocks in (a)

The quadtree and bintree decompose a region into equal-sized parts. Kanatani [Kana85] suggests using splitting rules based on the Fibonacci sequence of numbers. The Fibonacci numbers consist of the sequence of numbers f_i that satisfy the relation $f_i = f_{i-1} + f_{i-2}$, with $f_0 = 1$ and $f_1 = 1$. We can try to devise both quadtree and bintree splitting rules based on such a sequence. Generally for a decomposition scheme to be useful in geometric applications, it must have pixel-sized squares (i.e., 1×1) as the primitive tiles. At first glance, it appears that the Fibonacci sequence gives quite a bit of leeway in deciding on a splitting sequence and on the sizes of the regions corresponding to the subtrees and the primitive tiles.

One possible quadtree splitting rule is to restrict all shapes to squares with sides whose lengths are Fibonacci numbers. Clearly not all the shapes can be squares since we cannot aggregate these squares into larger squares that obey this rule. Another possibility is to restrict the shapes to rectangles the length of whose sides are either equal Fibonacci numbers or are successive Fibonacci numbers (see Exercise 1.26). We term this condition the *2-d Fibonacci condition*.

In this discussion, we have assumed splitting rules that ensure that vertical subdivision lines at the same level are colinear as well as for horizontal lines at the same level. For example, when using a quadtree splitting rule, the vertical lines that subdivide the NW and SW quadrants are colinear, as well as for the horizontal lines that subdivide the NW and NE quadrants. An alternative is to relax the colinearity restriction; however, the sides of the shapes must still satisfy the 2-d Fibonacci condition (see Exercise 1.27).

As can be seen in Exercises 1.26 and 1.27, neither a quadtree nor a bintree can

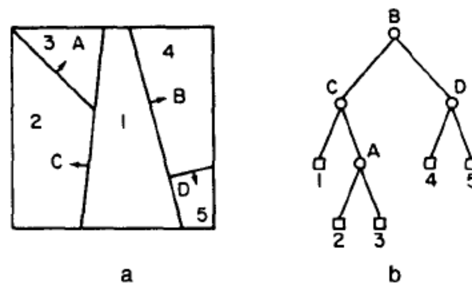


Figure 1.19 (a) An arbitrary space decomposition and (b) its BSP tree. The arrows indicate the direction of the positive halfspaces.

be used by itself as a basis for Fibonacci-based space decomposition; however, a combination of the two structures could be used. When the lengths of the sides of a rectangle are equal, the rectangle is split into four rectangles such that the lengths of the sides satisfy the 2-d Fibonacci condition. When the lengths of the sides of a rectangle are not equal, the rectangle is split into two rectangles with the split along a line (an axis) parallel to the shorter (longer) of the two sides. Interestingly the dimensions of the A-series of European paper are based on a Fibonacci sequence—that is, the elements of the series are of dimension $f_i \times f_{i-1}$ multiplied by an appropriate scale factor.

Another variation on the bintree idea, termed *adaptive hierarchical coding* (*AHC*), is proposed by Cohen, Landy, and Pavel [Cohe85b]. In this case, the image is again split into two equal-sized parts at each stage, but there is no need to alternate between the x and y coordinates. The decision as to the coordinate on which to partition depends on the image. This technique may require some work to get the optimal partition from the point of view of a minimum number of nodes (see Exercise 1.29).

An even more general variation on the bintree is the *BSP* tree of Fuchs, Kedem, and Naylor [Fuch80, Fuch83]. Its variants are used in some hidden-surface elimination algorithms (see Section 7.1.5 of [Same90b]) and in some implementations of beam tracing (see Section 7.3 of [Same90b]). It is applicable to data of arbitrary dimension, although here it is explained in the context of two-dimensional data. At each subdivision stage, the image is subdivided into two parts of arbitrary size. Note that successive subdivision lines need be neither orthogonal nor parallel. Therefore the resulting decomposition consists of arbitrarily shaped convex polygons.

The BSP tree is a binary tree. To be able to assign regions to the left and right subtrees, we associate a direction with each subdivision line. In particular, the subdivision lines are treated as separators between two halfspaces.[9] Let the line have the

[9] A (linear) *halfspace* in d-space is defined by the inequality $\sum_{i=0}^{d} a_i \cdot x_i \geq 0$ on the $d+1$ homogeneous coordinates ($x_0 = 1$). The halfspace is represented by a column vector a. In vector notation, the inequality is written as $a \cdot x \geq 0$. In the case of equality, it defines a hyperplane with a as its normal. It is important to note that halfspaces are volume, not boundary, elements.

 Unified Patents Exhibit 1005 App'x A-N

equation $a \cdot x + b \cdot y + c = 0$. We say that the right subtree is the 'positive' side and contains all subdivision lines formed by separators that satisfy $a \cdot x + b \cdot y + c \geq 0$. Similarly we say that the left subtree is 'negative' and contains all subdivision lines formed by separators that satisfy $a \cdot x + b \cdot y + c < 0$. As an example, consider Figure 1.19a, which is an arbitrary space decomposition whose BSP tree is given in Figure 1.19b. Notice the use of arrows to indicate the direction of the positive halfspaces.

Exercises

1.17. Given a $[6^3]$ tiling such that each side of an atomic tile has a unit length, compute the three adjacency distances from the centroid of an atomic tile.
1.18. Repeat Exercise 1.17 for $[3^6]$ and $[4^4]$, again assuming that each side of an atomic tile has a unit length.
1.19. Suppose that you are given an image in the form of a binary array of pixels. The result is a square grid. How can you view this grid as a hexagonal grid?
1.20. Show how the property sphere data structure can be used to model the earth. In particular, discuss how to represent landmass features, such as mountain ranges and crevices.
1.21 Suppose that you use an icosahedron to model spherical data. Initially there are 20 faces. How many faces are there after the first level of decomposition when $n = 2$? $n = \sqrt{3}$?
1.22. What is the ratio of leaf nodes to nonleaf nodes in a bintree for a d-dimensional image?
1.23. What is a lower bound on the ratio of leaf nodes in a bintree to that in a quadtree for a d-dimensional image? What is an upper bound? What is the average?
1.24. Is it true that the total number of nodes in a bintree is always less than that in the corresponding quadtree?
1.25. The Fibonacci numbers are defined by the relation $f_n = f_{n-1} + f_{n-2}$. Devise a two-dimensional analog of this relation to correspond to a splitting rule that would have to be satisfied in a Fibonacci-based space decomposition that yields four parts. Generalize this result to n dimensions.
1.26. Give a counterexample to the use of a quadtree splitting rule in a Fibonacci-based space decomposition.
1.27. Give a counterexample to the use of a bintree splitting rule in a Fibonacci-based space decomposition.
1.28. Suppose that you use the combination quadtree-bintree approach to a Fibonacci-based space decomposition. Prove that any image such that the lengths of its sides satisfy the 2-d Fibonacci condition can be decomposed into subimages whose sides obey this property and with a primitive tile of size 1×1.
1.29. Suppose that you use the AHC method. How many different rectangles and positions must be examined in building such a structure for a $2^n \times 2^n$ image?

1.4.2 Nonpolygonal Tilings

In the previous section we focused on space decompositions based on polygonal tiles. This is the prevalent method in use today. For certain applications, however, the use of polygonal tiles can lead to problems. For example, suppose that we have a decomposition based on square tiles. In this case, as the resolution is increased, the area of the approximated region approaches the true value of the area; however, this is not

true for a boundary measure such as the perimeter. To see this, consider a quadtree approximation of an isosceles right triangle where the ratio of the approximated perimeter to the true perimeter is $4/(2+\sqrt{2})$ (see Exercise 1.30). Other problems include the discontinuity of the normals to the boundaries of adjacent tiles.

There are a number of ways of attempting to overcome these problems. The *hierarchical probe model* of Chen [Chen85b] is an approach based on treating space as a polar plane and recursively decomposing it into sectors. We say that each sector consists of an origin, two sides (labeled 1 and 2 corresponding to the order in which they are encountered when proceeding in a counterclockwise direction), and an arc. The points at which the sides of the sector intersect (or touch) the object are called *contact points*. (ρ,θ) denotes a point in the polar plane. Let (ρ_i, θ_i) be the contact point with the maximum value of ρ in direction θ_i. Each sector represents a region bounded by the points (0,0), (ρ_1,θ_1), and (ρ_2,θ_2), where $\theta_1 = 2k\pi/2^n$ and $\theta_2 = \theta_1 + 2\pi/2^n$ such that k and n are nonnegative integers ($k < 2^n$). The arc between the two nonorigin contact points (ρ_1,θ_1) and (ρ_2,θ_2) of a sector is approximated by the linear parametric equations ($0 \le t \le 1$):

$$\rho(t) = \rho_1 + (\rho_2 - \rho_1)\cdot t \qquad \theta(t) = \theta_1 + (\theta_2 - \theta_1)\cdot t.$$

Note that the interpolation curves are arcs of spirals due to the linear relation between ρ and θ.

The *sector tree* is a binary tree that represents the result of recursively subdividing sectors in the polar plane into two sectors of equal angular intervals. Thus the recursive decomposition is only with respect to θ, not ρ. The decomposition stops whenever the approximation of a part of an object by a sector is deemed to be adequate. The computation of the stopping condition is implementation dependent. For example, it can be the maximum deviation in the value of ρ between a point on the boundary and the corresponding point (i.e., at the same value of θ) on the approximating arc. Initially the universe is the interval $[0,2\pi)$.

In the presentation, we assume that the origin of the polar plane is contained within the object. See Exercise 1.36 for a discussion of how to represent an object that does not contain the origin of the polar plane. The simplest case arises when the object is convex. The result is a binary tree where each leaf node represents a sector and contains the contact points of its corresponding arc. For example, consider the object in Figure 1.20. The construction of its sector tree approximation is shown in

Figure 1.20 Example convex object

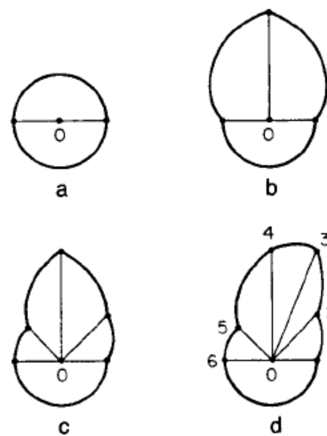


Figure 1.21 Successive sector tree approximations for the object of Figure 1.20: (a) π intervals, (b) π/2 intervals, (c) π/4 intervals, (d) π/8 intervals

Figures 1.21a–d. The final binary tree is given in Figure 1.22 with interval endpoints labeled according to Figure 1.21d.

The situation is more complex when the object is not convex. This means that each side of a sector may intersect the boundary of the object at an arbitrary, and possibly different, number of contact points. In the following, each sector will be seen to consist of a set of alternating regions within and outside the object. These regions are three-sided or four-sided and have at least one side that is colinear with a side of the sector. The discussion is illustrated with the object of Figure 1.23a whose sector tree decomposition is given in Figure 1.23b. The final binary tree is given in Figure 1.24. A better indication of the quality of the approximation can be seen by examining Figure 1.23c, which contains an overlay of Figures 1.23a and 1.23b.

When the boundary of the object intersects a sector at two successive contact points, say P and Q, that lie on the same side, say S, of the sector, then the region

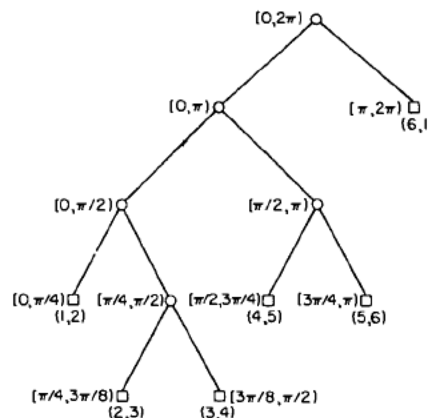


Figure 1.22 Binary tree representation of the sector tree of Figure 1.20

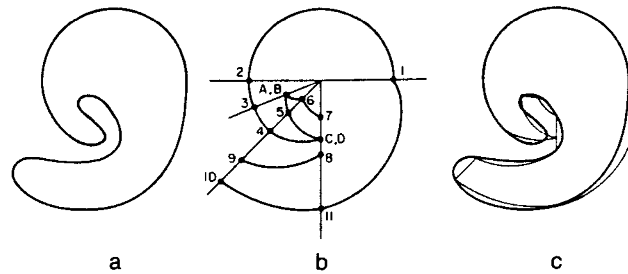


Figure 1.23 (a) Example object, (b) its sector tree description, and (c) a comparison of the sector tree approximation (thin lines) with the original object (thick lines). Note the creation of a hole corresponding to the region formed by points A, B, 6, 7, C, D, and 5

bounded by S and PQ must be approximated. Without loss of generality, assume that the region is inside the object. There are two choices. An inner approximation ignores the region by treating the segment of S between P and Q as part of the approximated boundary (e.g., the region between points 9 and 10 in sector $[9\pi/8, 5\pi/4)$ in Figure 1.23b).

An outer approximation inserts two identical contact points, say R and T, on the other side of the sector and then approximates the region by the three-sided region formed by the segment of S between P and Q and the spiral arc approximations of PR and QT. The value of R (and hence T) is equal to the average of the value of ρ at P and Q. For example, the region between points 4 and 5 in sector $[5\pi/4, 3\pi/2)$ in Figure 1.23b is approximated by the region formed with points C and D.

Of course, the same approximation process is applied to the part of the region outside the object. In Figure 1.23b, we have an inner approximation for the region between points 7 and 8 in sector $[3\pi/2, 2\pi)$, and an outer approximation for the region between points 5 and 6 in sector $[9\pi/8, 5\pi/4)$, by virtue of the introduction of points A and B.

One of the problems with the sector tree is that its use can lead to the creation of holes that do not exist in the original object. This situation arises when the decomposition is not carried out to a level of sufficient depth. For example, consider Figure 1.23b, which has a hole bounded by the arcs formed by points A, B, 6, 7, C, D, and 5. This is a result of the inner approximation for the region between points 7 and 8 in sector $[3\pi/2, 2\pi)$ and an outer approximation for the region between points 4 and 5 in sector $[5\pi/4, 3\pi/2)$. This situation can be resolved by further decomposition in either or both of sectors $[3\pi/2, 2\pi)$ and $[5\pi/4, 3\pi/2)$.

The result of the approximation process is that each sector consists of a collection of three-sided and four-sided regions that approximate the part of the object contained in the sector. This collection is stored in the leaf node of the sector tree as a list of pairs of points in the polar plane. It is interesting to observe that the boundaries of the interpolated regions are not stored explicitly in the tree. Instead each pair of points corresponds to the boundary of a region. Since the origin of the polar plane is within the object, an odd number of pairs of points is associated with each leaf node. For

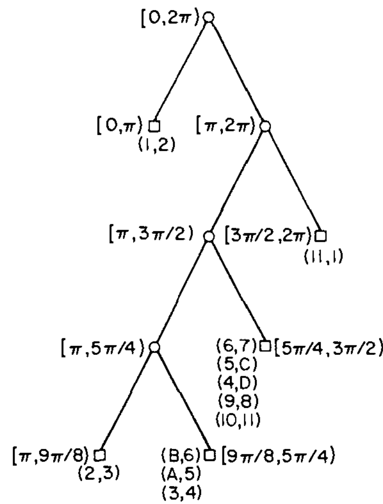


Figure 1.24 Binary tree representation of the sector tree of Figure 1.23

example, consider the leaf node in Figure 1.24 corresponding to the sector $[5\pi/4, 3\pi/2)$. The first pair, together with the origin, defines the first region (e.g., (6,7)). The next two pairs of points define the second region (e.g., (5,C) and (4,D)), with each successive two pairs of points defining the remaining regions.

The sector tree is a partial polar decomposition, as the subdivision process is based only on the value of θ. A total polar decomposition would partition the polar plane on the basis of both ρ and θ. The result is analogous to a quadtree, and it is termed a *polar quadtree*. There are a number of possible rules for the decomposition process (see Exercise 1.42). For example, consider a decomposition that recursively halves both ρ and θ at each level. In general, the polar quadtree is a variant of a maximal block representation. As in the sector tree, the blocks are disjoint. Unlike the sector tree, blocks in the polar quadtree do have standard sizes. In particular, all blocks in the polar quadtree are either three sided (i.e., sectors) or four sided (i.e., quadrilaterals, two of whose sides are arcs). Thus the sides of polar quadtree blocks are not based on interpolation.

The primary motivation for presenting the sector tree is to show that space decompositions could also be based on nonpolygonal tiles. In the rest of this book the primary concern is with space decompositions based on rectangles (especially squares) and showing how a number of operations can be performed when they serve as the underlying representation. The techniques are quite general and can be applied to most space decomposition methods. Thus the sector tree is not discussed further except in the context of its adaptation to the representation of three-dimensional data (see Section 5.6). Nevertheless, the following contains a brief mention of some of the operations to which the sector tree lends itself.

Set operations such as union and intersection are straightforward. Scaling is trivial as the sector tree need not be modified; all values of ρ are interpreted as scaled

by the appropriate scale factor. The number of nodes in a sector tree is dependent on its orientation—that is, on the points chosen as the origin and the contact point chosen to serve as $(\rho,0)$. Rotation is not so simple; it cannot be implemented by simply rearranging pointers (but see Exercise 1.40). Translation is computationally expensive since the change in the relative position of the object with respect to the origin means that the entire sector tree must be reconstructed.

Exercises

1.30. Prove that for an isosceles right triangle represented by a region quadtree, the ratio of the approximated perimeter to the true perimeter is $4/(2+\sqrt{2})$.

1.31. Repeat Exercise 1.30 for a circle (i.e., find the ratio).

1.32. When the objects have linear sides, polygonal tiles are superior. How would you use the sector tree decomposition method with polygonal tiles?

1.33. In the discussion of the situation arising when the boundary of the object intersects a sector at two successive contact points, say P and Q, that lie on the same side, say S, of the sector, we assumed that the region bounded by S and PQ was inside the object. Suppose that this region is outside the object. How does this affect the inner and outer approximations?

1.34. Can you traverse the boundary of an object represented by a sector tree by visiting each leaf node just once?

1.35. When using a sector tree, how would you handle the situation that the boundary of the object just touches the side of a sector without crossing it (i.e., a tangent if the boundary is differentiable)?

1.36. How would you use a sector tree to represent an object that does not contain the origin of the polar plane?

1.37. The outer approximation used in building a sector tree always yields a three-sided region. Two of the sides are arcs of spirals with respect to a common origin. This implies a sharp discontinuity of the derivative at the point at which they meet. Can you devise a way to smoothe this discontinuity?

1.38. Does the inner approximation used in building a sector tree always underestimate the area? Similarly does the outer approximation always overestimate the area?

1.39. Compare the inner and outer approximations used in building a sector tree. Is there ever a reason for the outer approximation to be preferred over the inner approximations (or vice-versa)?

1.40. Define a complete sector tree in an analogous manner to a complete binary tree—that is, all leaf nodes are at the same level, say n. Prove that a complete sector tree is invariant under rotation in multiples of $2\pi/2^n$.

1.41. Write an algorithm to trace the boundary of an object represented by a sector tree.

1.42. Suppose that it is desired to decompose space into nonpolygonal shapes. Develop a quadtree-like data structure based on polar coordinates (i.e., ρ and θ). Investigate different splitting rules for polar quadtrees. In particular, you do not need to alternate the splits—that is, you could split on ρ several times in a row, and so on. This technique is used in the adaptive k–d tree [Frie77] (see Section 2.4.1) by decomposing the quartering process into two splitting operations—one for the x coordinate and one for the y coordinate. What are the possible shapes for the quadrants of such trees (e.g., a torus, doughnut, wheels with spokes)?

1.5 SPACE REQUIREMENTS

The primary motivation for the development of the quadtree was the desire to reduce the amount of space necessary to store data through the use of aggregation of homogeneous blocks. As we will see in subsequent chapters, an important by-product of this aggregation is the reduction of the execution time of a number of operations (e.g., connected component labeling, component counting). However, a quadtree implementation does have overhead in terms of the nonleaf nodes. For an image with B and W black and white blocks, respectively, $4 \cdot (B + W)/3$ nodes are required. In contrast, a binary array representation of a $2^n \times 2^n$ image requires only 2^{2n} bits; however, this quantity grows quite quickly. Furthermore, if the amount of aggregation is minimal (e.g., a checkerboard image), the quadtree is not very efficient.

The overhead for the nonleaf nodes can be reduced at times by using a pointer-less representation. Pointer-less representations can be grouped into two categories. The first, termed a *DF-expression*, represents the quadtree as a traversal of its constituent nodes [Kawa80a]. For example, letting 'B', 'W', and 'G' correspond to black, white, and gray nodes, respectively, and assuming a traversal in the order NW, NE, SW, and SE, the quadtree of Figure 1.1 would be represented by GWGWWBBGWGW BBBWBGBBGBBBWW.

The second approach treats the quadtree as a collection of the leaf nodes comprising it. Each node is represented by a pair of numbers [Garg82c]. The first number is the level of the tree at which the node is located. The second number is termed a *locational code*. It is formed by a concatenation of base 4 digits corresponding to directional codes that locate the node along a path from the root of the quadtree. The directional codes take on the values 0, 1, 2, 3 corresponding to quadrants NW, NE, SW, SE, respectively. For example, node 15 in Figure 1.1 is represented by the pair of numbers (0,320), which is decoded as follows. The base 4 locational code is 320. The pair denotes a node at level 0 that is reached by a sequence of transitions, SE, SW, and NW, starting at the root. A quadtree representation based on the use of locational codes is called *linear quadtree* by Gargantini [Garg82a, Garg82c] (because the addresses are keys in a linear list of nodes). Pointer-less representations are discussed in greater detail in Chapter 2 of [Same90b].

The worst case for a quadtree of a given depth in terms of storage requirements occurs when the region corresponds to a checkerboard pattern as in Figure 1.25. The amount of space required is obviously a function of the resolution (i.e., the number of levels in the quadtree), the size of the image (i.e., its perimeter), and its positioning in the grid within which it is embedded. As a simple example, Dyer [Dyer82] has shown that arbitrarily placing a square of size $2^m \times 2^m$ at any position in a $2^n \times 2^n$ image requires an average of $O(2^{m+2} + n - m)$ quadtree nodes. An alternative characterization of this result is that the average amount of space necessary is $O(p+n)$ where p is the perimeter (in pixel widths) of the block.

Dyer's $O(p+n)$ result for a square image is merely an instance of the earlier work of Hunter and Steiglitz [Hunt78, Hunt79a] who proved some fundamental theorems on the space requirements of images represented by quadtrees. In their

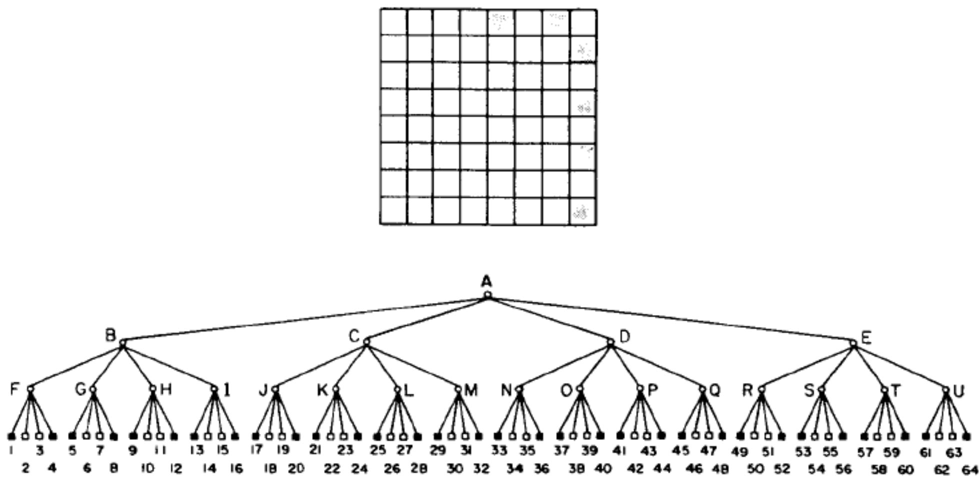


Figure 1.25 A checkerboard and its quadtree

studies, Hunter and Steiglitz used simple polygons (polygons with nonintersecting edges and without holes); however, these theorems have been observed to hold in arbitrary images (see [Rose82b] for empirical results in a cartographic environment).

In Hunter and Steiglitz's formulation, a polygon is represented by a three-color variant of the quadtree. In essence, there are three types of nodes: interior, boundary, and exterior. A node is said to be of type *boundary* if an edge of the polygon passes through it. *Interior* and *exterior* nodes correspond to areas within, and outside, respectively, the polygon and can be merged to yield larger nodes. The resulting quadtree is analogous to the MX quadtree representation of point data described below (for more details, see Section 2.6.1), and this term will be used to describe it. In particular, boundary nodes are analogous to black nodes, while interior and exterior nodes are analogous to white nodes.

Figure 1.26 illustrates a sample polygon and its MX quadtree. One disadvantage of the MX quadtree representation for polygonal lines is that a width is associated with them, whereas in a purely technical sense these lines have a width of zero. Also shifting operations may result in information loss. (For more appropriate representations of polygonal lines, see Chapter 4.)

An upper bound on the number of nodes in such a representation of a polygon can be obtained in the following manner. First, we observe that a curve of length $d + \varepsilon$ ($\varepsilon > 0$) can intersect at most six squares of side width d. Now consider a polygon, say G, having perimeter p, that is embedded in a grid of squares each of side width d. Mark the points at which G enters and exits each square. Choose one of these points, say P, as a starting point for a decomposition of G into a sequence of curves. Define the first curve in G to be the one extending from P until six squares have been intersected and a crossing is made into a different seventh square. This is the starting point for another curve in G that intersects six new squares, not counting those intersected by any previous curve.

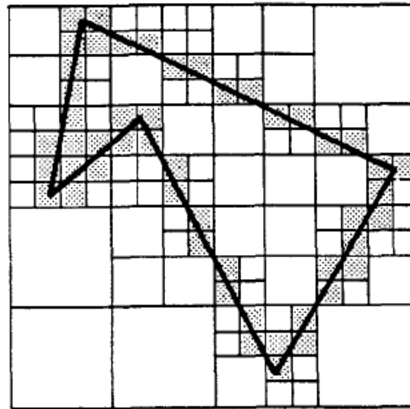


Figure 1.26 Hunter and Steiglitz's quadtree representation of a polygon

We now decompose G into a series of such curves. Since each curve adds at most six new squares and has length of at least d, we see that a polygon with perimeter p cannot intersect more than $6 \cdot \lceil p/d \rceil$ squares. Given a quadtree with a root at level n (i.e., the grid of squares is of width 2^n), at level i each square is of width 2^i. Therefore polygon G cannot intersect more than $B(i) = 6 \cdot \lceil p/2^i \rceil$ quadrants at level i. Recall that our goal is to derive an upper bound on the total number of nodes. This bound is attained when each boundary node at level i has three brother nodes that are not intersected. Of course, only boundary nodes can have sons, and thus no more than $B(i)$ nodes at level i have sons. Since each node at level i is a son of a node at level $i+1$, there are at most $4 \cdot B(i+1)$ nodes at level i. Summing up over n levels (accounting for a root node at level n and four sons), we find that the total number of nodes in the tree is bounded by

$$
\begin{aligned}
& 1 + 4 + \sum_{i=0}^{n-2} 4 \cdot B(i+1) \\
& \leq 5 + 24 \cdot \sum_{i=0}^{n-2} \left\lceil \frac{p}{2^{i+1}} \right\rceil \\
& \leq 5 + 24 \cdot \sum_{i=0}^{n-2} \left(1 + \frac{p}{2^{i+1}}\right) \\
& \leq 5 + 24 \cdot (n-1) + 24 \cdot p \cdot \sum_{i=0}^{n-2} \frac{1}{2^{i+1}} \\
& \leq 24 \cdot n - 19 + 24 \cdot p.
\end{aligned}
$$

Therefore, we have proved:

> **Theorem 1.1** The quadtree corresponding to a polygon with perimeter p embedded in a $2^n \times 2^n$ image has a maximum of $24 \cdot n - 19 + 24 \cdot p$ (i.e., $O(p+n)$) nodes. □

The proof of Theorem 1.1 is based on a decomposition of the polygon into a sequence of curves, each of which intersects at most six squares. This bound can be tightened by examining patterns of squares to obtain minimum lengths and corresponding ratios of possible squares per unit length. For example, observe that once a curve intersects six squares, the next curve of length d in the sequence can intersect at most two new squares. In contrast, it is easy to construct a sequence of curves of length $d + \varepsilon$ ($\varepsilon > 0$) such that almost each curve intersects two squares of side length d. Such a construction leads to an upper bound of the form $a \cdot n + b + 8 \cdot p$ where a and b are constants (see Exercise 1.48). Hunter and Steiglitz use a slightly different construction to obtain a bound of $16 \cdot n - 11 + 16 \cdot p$ (see Exercise 1.49).

Nevertheless, the bound of Theorem 1.1 is attainable as demonstrated by the following examples. First, consider a square of side width 2 that consists of the central four squares in a $2^n \times 2^n$ image (see Figure 1.27). Its quadtree has $16 \cdot n - 11$ nodes (see Exercise 1.50). Second, consider a curve that follows a vertical line through the center of a $2^n \times 2^n$ image. Now, make it a bit longer by making it intersect all of the pixels on either side of the vertical line (see Figure 1.28). As n increases, the total number of nodes in the quadtree approaches $8 \cdot p$ where $p = 2^n$ (see Exercise 1.51). A polygon having a number of nodes approaching $8 \cdot p$ can be constructed in a similar manner by approximating a square in the center of the image whose side is one-fourth the side of the image (see Exercise 1.52). In fact, it has been shown by Hunter [Hunt78] that $O(p+n)$ is a least upper bound on the number of nodes in a quadtree corresponding to a polygon (see Exercise 1.53).

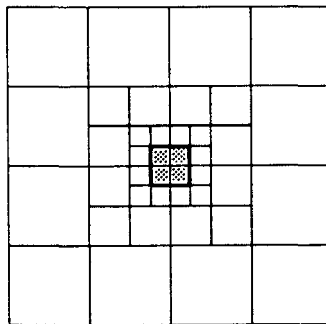


Figure 1.27 Example quadtree with $16 \cdot n - 11$ nodes

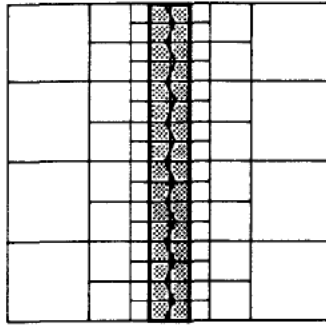


Figure 1.28 Example quadtree with approximately $8 \cdot p$ nodes

Theorem 1.1 can be recast by measuring the perimeter p in terms of the length of a side of the image in which the polygon is embedded—i.e., for a $2^n \times 2^n$ image $p = p' \cdot 2^n$. Thus the value of the perimeter no longer depends on the resolution of the image. Restating Theorem 1.1 in terms of p' results in a quadtree having $O(p' \cdot 2^n + n)$ nodes. This leads to the following important corollary:

Corollary 1.1 The maximum number of nodes in a quadtree corresponding to an image is directly proportional to the resolution of the image. □

The significance of Corollary 1.1 is that when using quadtrees, increasing the image resolution leads to a linear growth in the number of nodes. This is in contrast to the binary array representation where doubling the resolution leads to a quadrupling of the number of pixels.

Since in most practical cases the perimeter, p, dominates the resolution, n, the results of Theorem 1.1 are usually interpreted as stating that the number of nodes in a quadtree is proportional to the perimeter of the regions contained therein.[10] Meagher [Meag80] has shown that this theorem also holds for three-dimensional data (i.e., for polyhedra represented by octrees) when the perimeter is replaced by the surface area. The perimeter and the surface area correspond to the size of the boundary of the polygon and polyhedron—that is, in two and three dimensions, respectively. In d dimensions this result can be stated as follows:

Theorem 1.2: The size of a d-dimensional quadtree of a d-dimensional polyhedron is proportional to the sum of the resolution and the size of the boundary of the object. □

---

[10] Of course, the storage used by runlength codes is also proportional to the perimeter of the regions. However, runlength codes do not facilitate access to different parts of the regions (i.e., they have poor spatial indexing properties).

Aside from their implications on the storage requirements, Theorems 1.1 and 1.2 also directly affect the analysis of the execution time of algorithms. In particular, most algorithms that execute on a quadtree representation of an image instead of an array representation have an execution time proportional to the number of blocks in the image rather than the number of pixels. In its most general case, this means that the application of a quadtree algorithm to a problem in d-dimensional space executes in time proportional to the analogous array-based algorithm in the $(d-1)$-dimensional space of the surface of the original d-dimensional image. Thus quadtrees are somewhat like dimension-reducing devices.

Theorem 1.2 assumes that the image consists of a polyhedron. Walsh [Wals85] lifts this restriction and obtains a weaker complexity bound. Assuming an image of resolution n and measuring the perimeter, say p, in terms of the number of border pixels, he proves that the total number of nodes in a d-dimensional quadtree is less than or equal to $4 \cdot n \cdot p$. Furthermore he shows that the number of black nodes is less than or equal to $(2^d - 1) \cdot n \cdot p / d$.

The complexity measures discussed above do not explicitly reflect the fact that the amount of space occupied by a quadtree corresponding to a region is extremely sensitive to its orientation (i.e., where it is partitioned). For example, in Dyer's experiment, the number of nodes required for the arbitrary placement of a square of size $2^m \times 2^m$ at any position in a $2^n \times 2^n$ image ranged between $4 \cdot (n-m) + 1$ and $4 \cdot p + 16 \cdot (n-m) - 27$, with the average being $O(p+n-m)$. Clearly shifting the image within the space in which it is embedded can reduce the total number of nodes. The problem of finding the optimal position for a quadtree can be decomposed into two parts. First, we must determine the optimal grid resolution and, second, the partition points.

Grosky and Jain [Gros83] have shown that for a region such that w is the maximum of its horizontal and vertical extent (measured in pixel widths) and $2^{n-1} < w \leq 2^n$, the optimal grid resolution is either n or $n+1$. In other words embedding the region in a larger area than $2^{n+1} \times 2^{n+1}$ and shifting it around will not result in fewer nodes. Using similar reasoning, it can be shown that translating a region by 2^k pixels in any direction does not change the number of black or white blocks of size less than $2^k \times 2^k$ [Li82].

Armed with the above results, Li, Grosky, and Jain [Li82] developed the following algorithm that treats the image as a binary array and finds the configuration of the region in the image so that its quadtree requires a minimum number of nodes. First, enlarge the image to be $2^{n+1} \times 2^{n+1}$, and place the region within it so that the region's northernmost and westernmost pixels are adjacent to the northern and western borders, respectively, of the image. Next apply successive translations to the image of magnitude power of two in the vertical, horizontal, and corner directions and keep count of the number of leaf nodes required. Initially 2^{2n+2} leaf nodes are necessary. The following is a more precise statement of the algorithm:

1. Attempt to translate the image by (x,y) where x and y correspond to unit translations in the horizontal and vertical directions, respectively. Each of x and y takes on the values 0 or 1.

2. For the result of each translation in step 1, construct a new array at one-half the resolution. Each entry in the new array corresponds to a 2×2 block in the translated array. For each entry in the new array that corresponds to a single color (not gray) 2×2 block in the translated array, decrement the leaf node count by 3.
3. Recursively apply steps 1 and 2 to each result of steps 1 and 2. This process stops when no single-color 2×2 block is found in step 2 (i.e., they are all gray) or if the new array is a 1×1 block. Record the total translation and the minimum leaf node count.

Step 2 makes use of the property that for a translation of 2^k, there is a need to check only if single-color blocks of size $2^k \times 2^k$ or more are formed. In fact, because of the recursion, at each step we check only for the formation of blocks of size $2^{k+1} \times 2^{k+1}$. Note that the algorithm tries every possible translation since any integer can be decomposed into a summation of powers of two (i.e., use its binary representation). In fact this is why a translation of (0,0) is part of step 1. Although the algorithm computes the positioning of the quadtree with the minimum number of leaf nodes, it is also the positioning of the quadtree with the minimum total number of nodes since the number of nonleaf nodes in a quadtree of T leaf nodes is $(T-1)/3$.

As an example of the algorithm, consider the region given in Figure 1.29a whose block decomposition is shown in Figure 1.29b. Its quadtree requires 52 leaf nodes. The first step is to enlarge the image, place the region in the upper left corner, and form the array (Figure 1.30). The optimal positioning is such that Figure 1.30 is shifted 7 units in the horizontal direction and 3 units in the vertical direction. This corresponds to a sequence of translations (1,1), (1,1), and (1,0). The intermediate translated arrays are shown in Figure 1.31. All gray nodes in the translated arrays are labeled with a 'G' while black nodes are shaded. The optimal quadtree contains 46 leaf nodes and is given in Figure 1.32.

Now let us trace the algorithm as it applies the optimal sequence of translations, in more detail. Initially the leaf node count is 256. A translation of (1,1) leads to Figure 1.31a where 58 of the array entries correspond to single-color 2×2 blocks in the translated array. The leaf node count is decremented by $58 \cdot 3 = 174$, resulting in

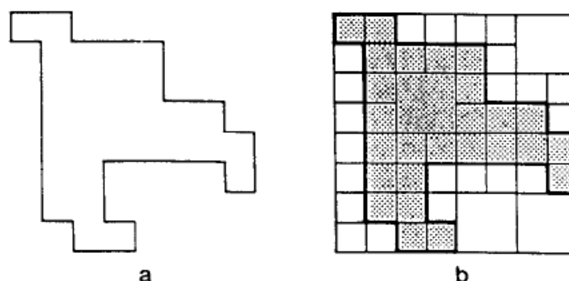


Figure 1.29 Example (a) image and (b) its block decomposition used to demonstrate the optimal positioning process

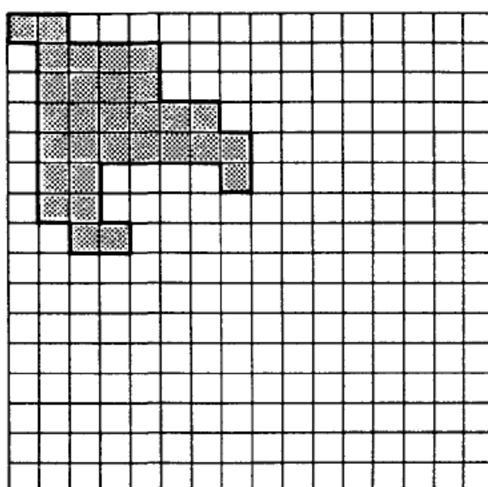


Figure 1.30 The array corresponding to the image in Figure 1.29 prior to the start of the optimal positioning process

82. The next translation of (1,1) leads to Figure 1.31b, where 11 of the array entries correspond to single-color 2×2 blocks. Therefore $11 \cdot 3 = 33$ is subtracted from 82, and the leaf node count is now 49. The final translation of (1,0) leads to Figure 1.31c, where only one of the array entries corresponds to a single-color 2×2 block in the translated array. Decrementing the leaf node count results in 46 nodes, and the process terminates. Of course, we have failed to describe the remaining $4^n - 3$ translations that were also attempted.

Despite trying all possible translations, the algorithm is quite efficient. The key is that for each translation, only the blocks whose motion can lead to space saving need to be considered. This is a direct consequence of the property that a translation of 2^k does not change the number of blocks of size less than $2^k \times 2^k$. For an image that has been enlarged to fit in a $2^{n+1} \times 2^{n+1}$ array, the algorithm will have a maximum depth of recursion of n. Since at each level of recursion we need an array at half the resolution of the previous level, the total amount of space required is $(4/3) \cdot 2^{2n+2}$.

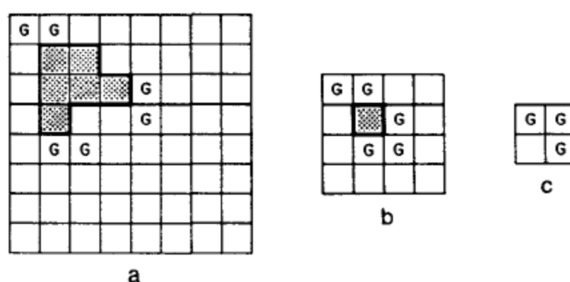


Figure 1.31 The successive translated arrays at half-resolution after application of (a) (1,1) and (b) (1,1), and (c) (1,0) to the original image array of Figure 1.30

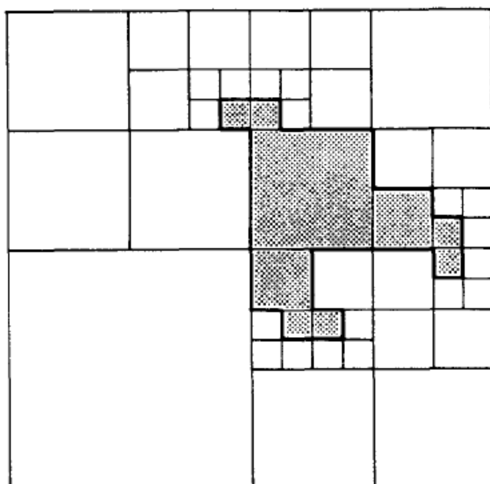


Figure 1.32 Optimal positioning of the quadtree of Figure 1.29

The basic computational task of the algorithm is to count 2×2 blocks of a single color. It can be shown that $4 \cdot n \cdot 2^{2n+2}$ array elements are examined in this process (see Exercise 1.63). Thus the algorithm uses $O(2^{2n})$ space and takes $O(n \cdot 2^{2n})$ time. Nevertheless experiments with typical images show that the algorithm has little effect (e.g., [Same84c]).

Exercises

1.43. Consider the arbitrary placement of a square of size $2^m \times 2^m$ at any position in a $2^n \times 2^n$ image. Prove that in the best case $4 \cdot (n-m) + 1$ nodes are required, while the worst case requires $4 \cdot p + 16 \cdot (n-m) - 27$ nodes. How many of these nodes are black and white, assuming that the square is black? Prove that on the average, the number of nodes that is required is $O(p+n-m)$.

1.44. What are the worst-case storage requirements of storing an arbitrary rectangle in a quadtree corresponding to a $2^n \times 2^n$ image? Give an example of the worst case and the number of nodes it requires.

1.45 Assume that the probability of a particular pixel's being black is one-half and likewise for being white. Given a $2^n \times 2^n$ image represented by a quadtree, what is the expected number of nodes, say $E(n)$, in the quadtree? Also compute the expected number of black, white, and gray nodes.

1.46 Suppose that instead of knowing the probability a particular pixel is black or white, we know the percentage of the total pixels in the image that are black. Given a $2^n \times 2^n$ image represented by a quadtree, what is the expected number of nodes in the quadtree?

1.47. The proof of Theorem 1.1 and the subsequent discussion raise the question of how N squares should be arranged so that each is intersected by a curve of minimum length extending to the outside of the squares on each end. Such a configuration leads to a minimal curve in the sense that it has a maximal ratio of squares to length. For which value of N is this ratio the smallest?

1.48. Try to prove that the upper bound of Theorem 1.1 can be tightened to be $a \cdot n + b + 8 \cdot p$ where a and b are constants.

1.49. Decompose the polygon used in the proof of Theorem 1.1 into a sequence of curves in the following manner. Mark the points where G enters and exits each square of side width d. Choose one of these points, say P, and define the first curve in G as extending from P until four squares have been intersected and a crossing is made into a different fifth square. This is the starting point for another curve in G that intersects four new squares, not counting those intersected by any previous curve. Prove that all of the curves, except for the last one, must be at least of length d. Using this result, prove that the upper bound on the number of nodes in the quadtree is $16 \cdot n - 11 + 16 \cdot p$.

1.50. Prove that the quadtree corresponding to a square of side width 2 consisting of the central four squares in a $2^n \times 2^n$ image has $16 \cdot n - 11$ nodes (see Figure 1.27).

1.51. Take a curve that follows a vertical line through the center of a $2^n \times 2^n$ image and lengthen it slightly by making it intersect all of the pixels on either side of the vertical line (see Figure 1.28). Prove that as n increases, the total number of nodes in the quadtree approaches $8 \cdot p$ where $p = 2^n$.

1.52. Using a technique analogous to that used in Exercise 1.51, construct a polygon of perimeter p by approximating a square in the center of the image whose side is one-fourth the side of the image. Prove that its quadtree has approximately $8 \cdot p$ nodes.

1.53. Prove that $O(p+n)$ is a least upper bound on the number of nodes in a quadtree corresponding to a polygon. Assume that $p \leq 2^{2n}$ (i.e., the number of pixels in the image). Equivalently the polygon boundary can touch all of the pixels in the most trivial way but can be no longer. Decompose your proof into two parts depending on whether p is greater than $4 \cdot n$.

1.54. Can you prove that for an arbitrary quadtree (not necessarily a polygon), the number of nodes doubles as the resolution is doubled?

1.55. Derive a result analogous to Theorem 1.1 for a three-dimensional polyhedron represented as an octree. In this case the perimeter corresponds to the surface area.

1.56. Prove Theorem 1.2.

1.57. Assuming an image of resolution n and measuring the perimeter, say p, in terms of the number of border pixels, prove that the total number of nodes in a d-dimensional quadtree is less than or equal to $4 \cdot n \cdot p$.

1.58. Assuming an image of resolution n and measuring the perimeter, say p, in terms of the number of border pixels, prove that the total number of black nodes in a d-dimensional quadtree is less than or equal to $(2^d - 1) \cdot n \cdot p / d$.

1.59. How tight are the bounds obtained in Exercises 1.57 and 1.58 for the number of nodes in a d-dimensional quadtree for an arbitrary region? Are they realizable?

1.60. Prove that for a region such that w is the maximum of its horizontal and vertical extent (measured in pixel widths) and $2^{n-1} < w \leq 2^n$, the optimal grid resolution is either n or $n+1$.

1.61. Prove that translating a region by 2^k pixels in any direction does not change the number of black or white blocks of size less than $2^k \times 2^k$.

1.62. Can you formally prove that the method described in the text does indeed yield the optimal quadtree?

1.63. Prove that $4 \cdot n \cdot 2^{2n+2}$ array elements are examined in the process of constructing the optimal quadtree.

1.64. How would you find the optimal bintree?

 Unified Patents Exhibit 1005 App'x A-N

US005263136A

United States Patent [19]

DeAguiar et al.

[11] **Patent Number: 5,263,136**

[45] **Date of Patent: Nov. 16, 1993**

[54] **SYSTEM FOR MANAGING TILED IMAGES USING MULTIPLE RESOLUTIONS**

[75] Inventors: **John R. DeAguiar**, Sebastopol; **Ross M. Larkin**, Rollings Hills, both of Calif.

[73] Assignee: **Optigraphics Corporation**, San Diego, Calif.

[21] Appl. No.: **694,416**

[22] Filed: **Apr. 30, 1991**

[51] **Int. Cl.**^{5} **G06F 15/20**

[52] **U.S. Cl.** **395/164**; 345/201

[58] **Field of Search** 395/162, 164, 166, 128–130; 340/798, 799; 358/452, 455

[56] **References Cited**

U.S. PATENT DOCUMENTS

Re. 31,200	4/1983	Sukonick et al.	395/162
4,878,183	10/1989	Ewart	395/128 X
4,920,504	4/1990	Sawada et al.	395/166
4,951,230	8/1990	Dalrymple et al.	395/166
5,020,003	5/1991	Moshenberg	395/164
5,150,462	9/1992	Takeda et al.	395/166

Primary Examiner—Dale M. Shaw
Assistant Examiner—Kee M. Tung
Attorney, Agent, or Firm—Knobbe, Martens, Olson & Bear

[57] **ABSTRACT**

An image memory management system for tiled images. The system defines an address space for a virtual memory that includes an image data cache and a disk. An image stack for each source image is stored as a full resolution image and a set of lower-resolution subimages. Each tile of an image may exist in one or more of five different states as follows: uncompressed and resident in the image data cache, compressed and resident in the image data cache, uncompressed and resident on disk, compressed and resident on disk and not loaded but re-creatable using data from higher-resolution image tiles.

17 Claims, 39 Drawing Sheets

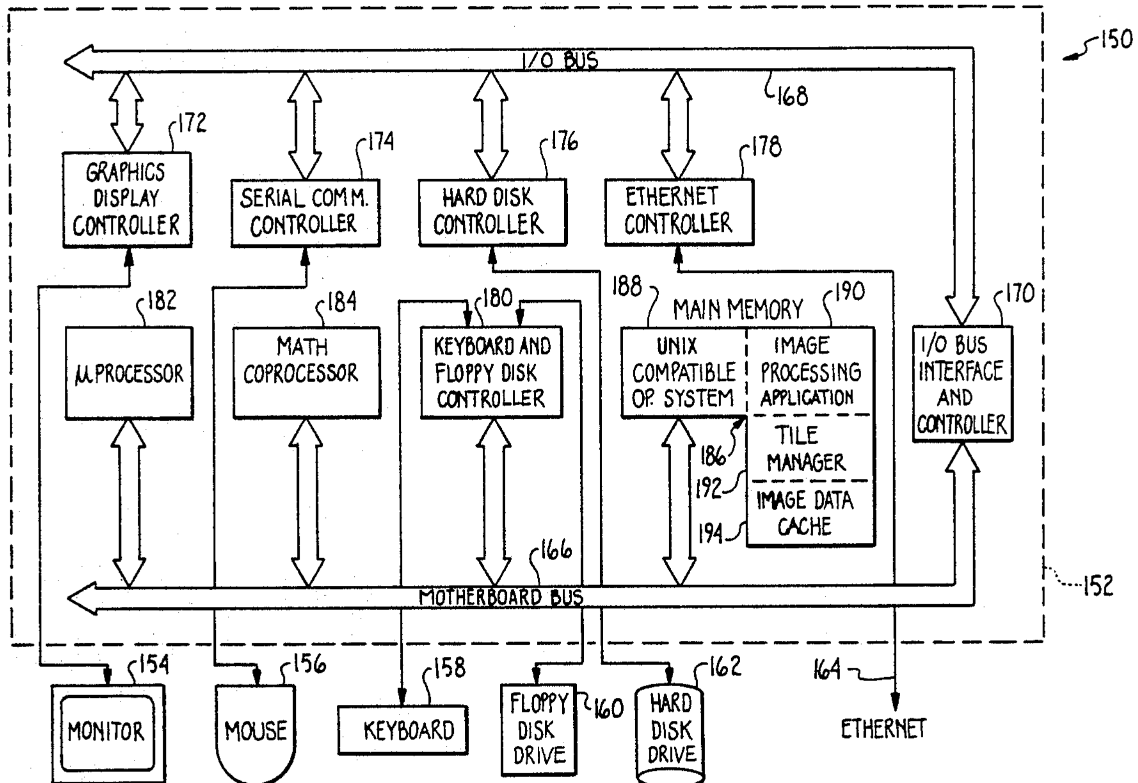
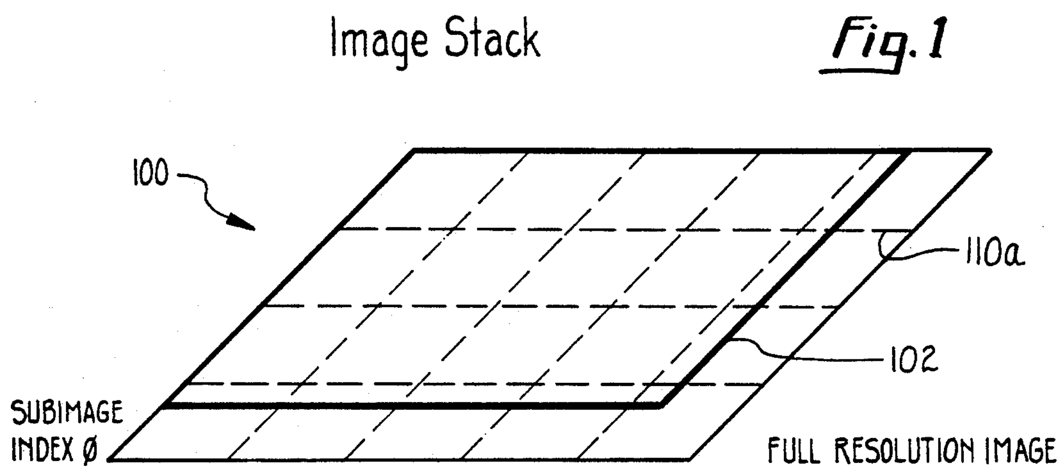
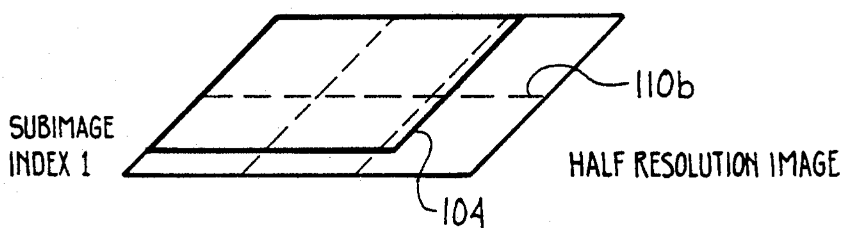
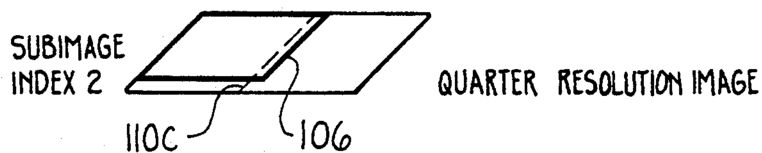


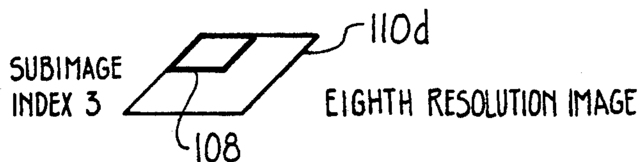
Image Stack

Fig. 1









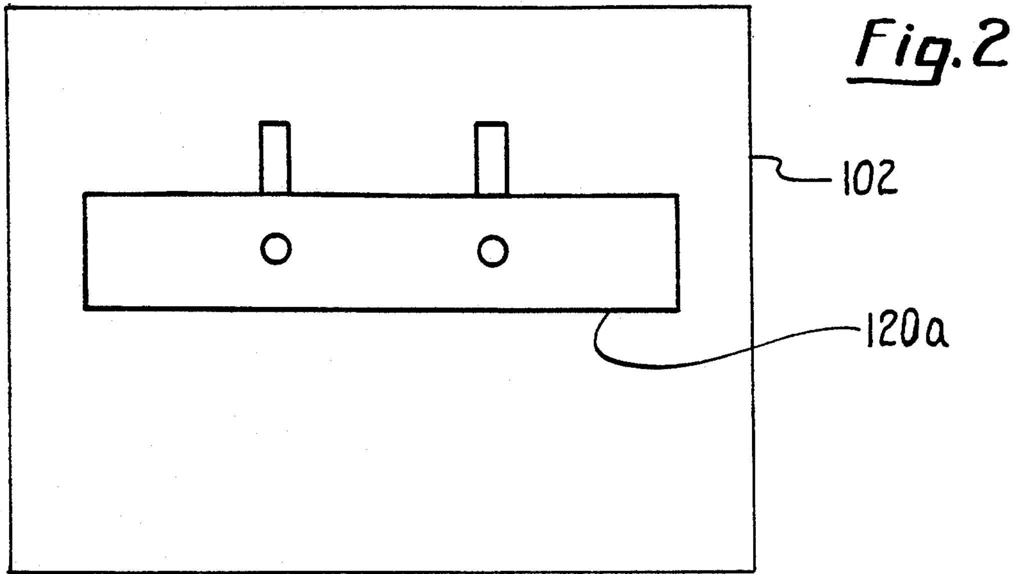


Fig. 2
102
120a

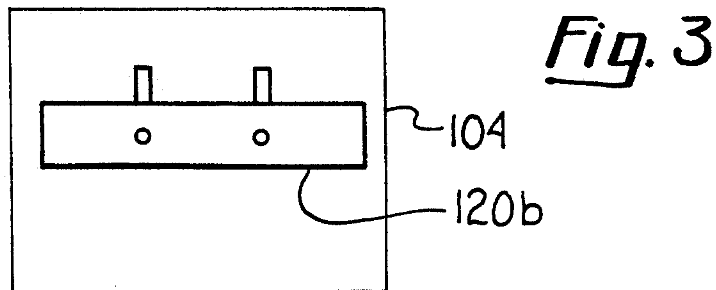


Fig. 3
104
120b

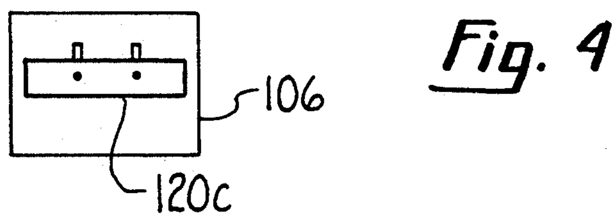
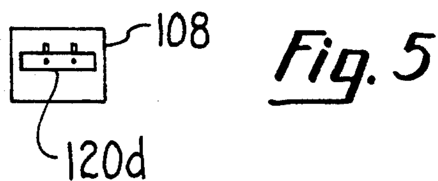
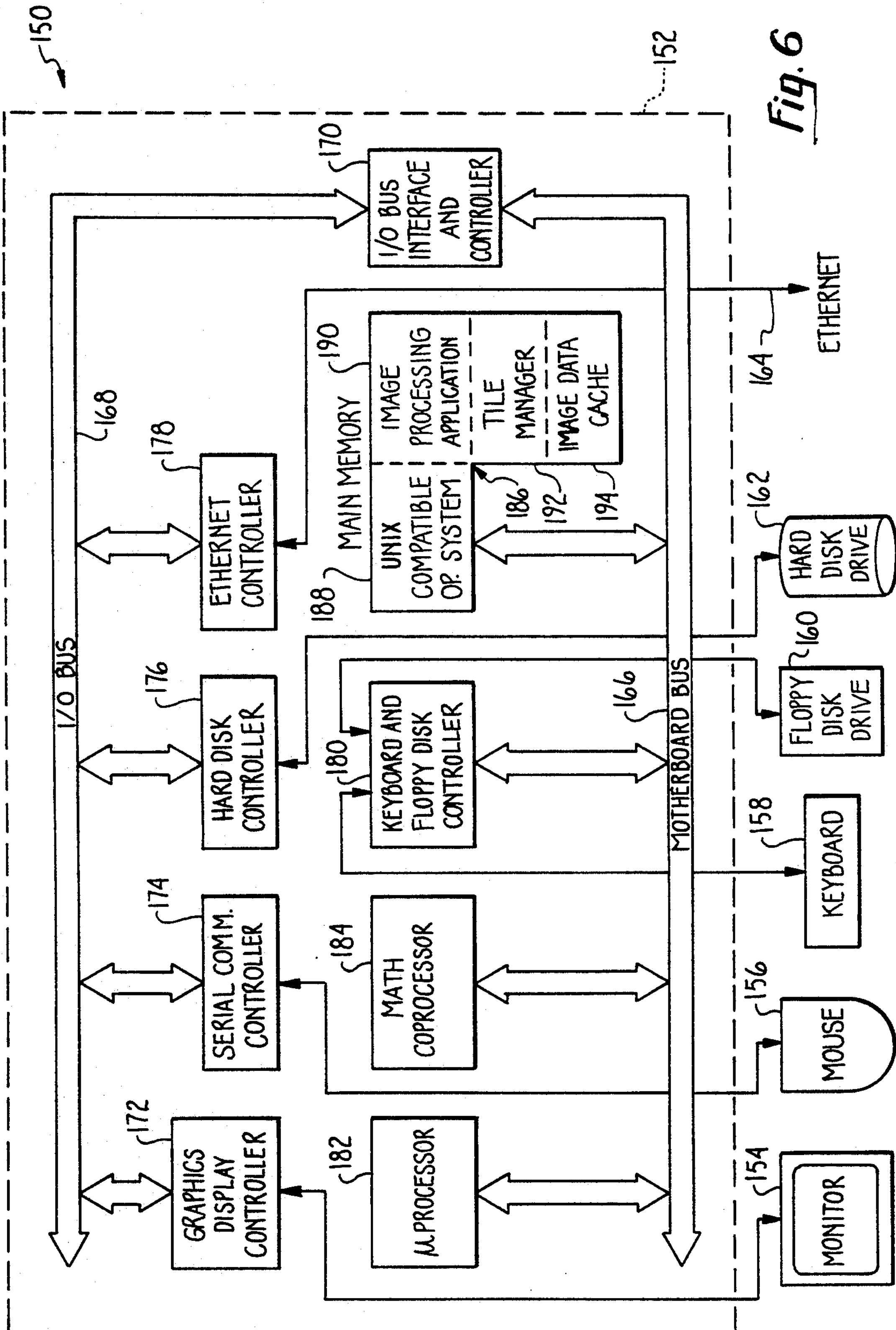


Fig. 4
106
120c


108
Fig. 5
120d


150
152
Fig. 6
170
I/O BUS INTERFACE AND CONTROLLER
ETHERNET
164
168
178
ETHERNET CONTROLLER
190
MAIN MEMORY
IMAGE PROCESSING APPLICATION
TILE MANAGER
IMAGE DATA CACHE
UNIX COMPATIBLE OP. SYSTEM
186
192
194
188
162
HARD DISK DRIVE
160
FLOPPY DISK DRIVE
I/O BUS
176
HARD DISK CONTROLLER
180
KEYBOARD AND FLOPPY DISK CONTROLLER
166
MOTHERBOARD BUS
158
KEYBOARD
174
SERIAL COMM. CONTROLLER
184
MATH COPROCESSOR
156
MOUSE
172
GRAPHICS DISPLAY CONTROLLER
182
µ.PROCESSOR
154
MONITOR

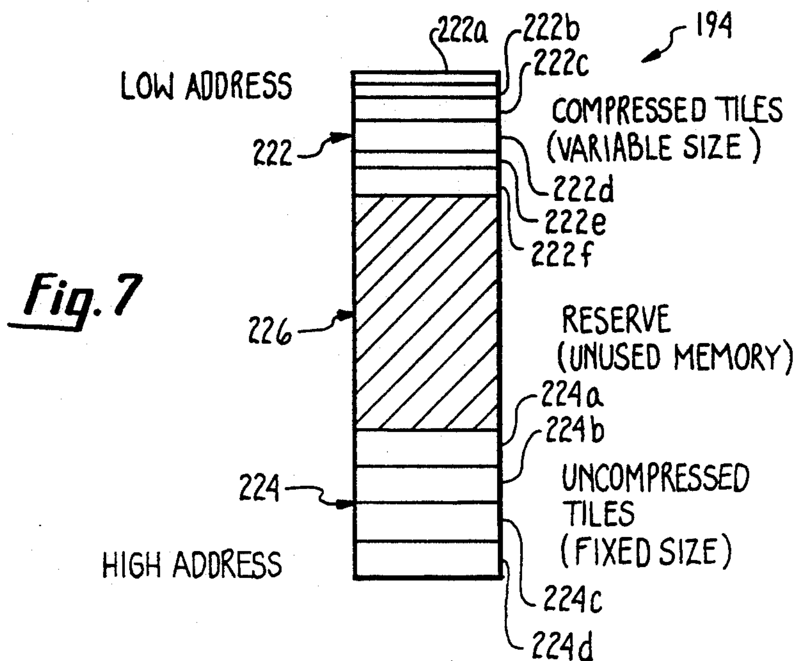


Fig. 7

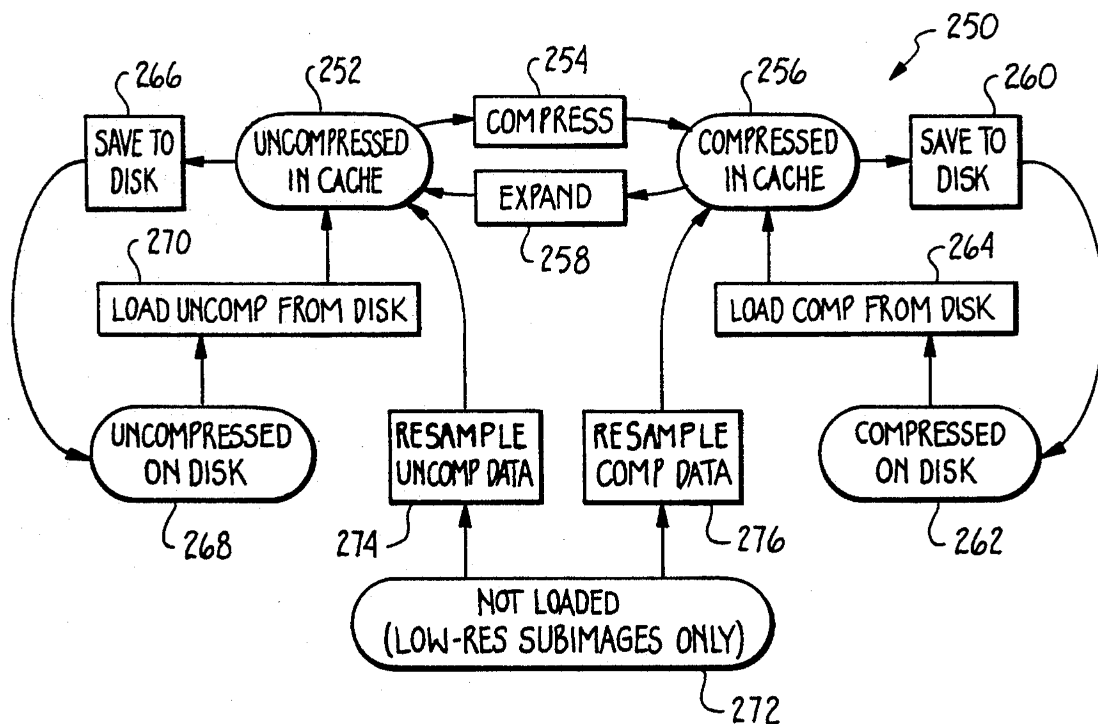


Fig. 8

Fig. 9A

Document Information Structure

300

SELF-REFERENCE TO DOCUMENT HANDLE 302	"OVERVIEWS INVALID" FLAG 304	CACHE IMAGE COMPRESSION ALGORITHM 306
IMAGE COLOR TYPE 308	BITS PER IMAGE PIXEL 310	TILE SIZE INFORMATION 312
NUMBER OF SUBIMAGES IN DOC 314	INPUT FILE INFO 316	OUTPUT FILE INFO 318
LIST OF SUBIMAGE HEADERS 320		

Fig. 9B

321

POINTER TO TILE HEADERS 322	POINTER TO TILE DIRECTORY 324	SUBIMAGE WIDTH AND HEIGHT 326
NUMBER OF TILE ROWS & COLS IN SUBIMAGE 328	IMAGE STACK INDEX OF THIS SUBIMAGE 330	PIXEL RESOLUTION OF THIS SUBIMAGE 332
⋮		

Fig. 10

Tile Header

350

POINTER TO DOCUMENT CONTAINING THIS TILE 352	INDEX OF SUBIMAGE CONTAINING THIS TILE 354	ROW AND COLUMN INDICES OF TILE 356

STATUS INFORMATION 358	PRESERVE COUNT 360
LOCATION OF UNCOMPRESSED IMAGE DATA IN CACHE MEMORY 362	LOCATION OF COMPRESSED IMAGE DATA IN CACHE MEMORY 364
LOCATION OF UNCOMPRESSED IMAGE DATA ON DISK 366	LOCATION OF COMPRESSED IMAGE DATA ON DISK 368
LINK TO NEXT LESS RECENTLY USED TILE 370	LINK TO NEXT MORE RECENTLY USED TILE 372
NUMBER OF BYTES OF EXPANDED DATA IN TILE 374	NUMBER OF BYTES OF COMPRESSED DATA IN TILE 376

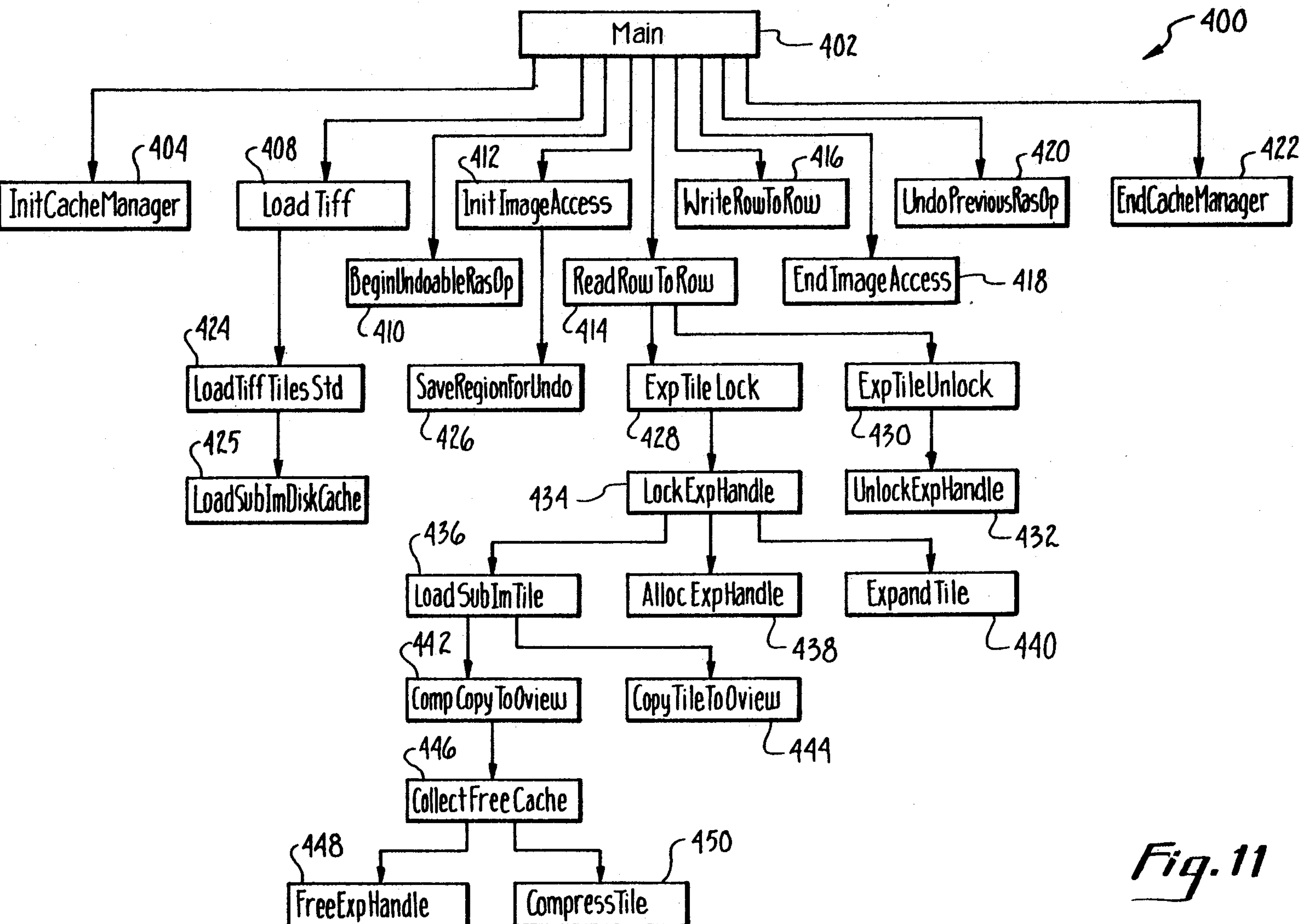


Fig. 11

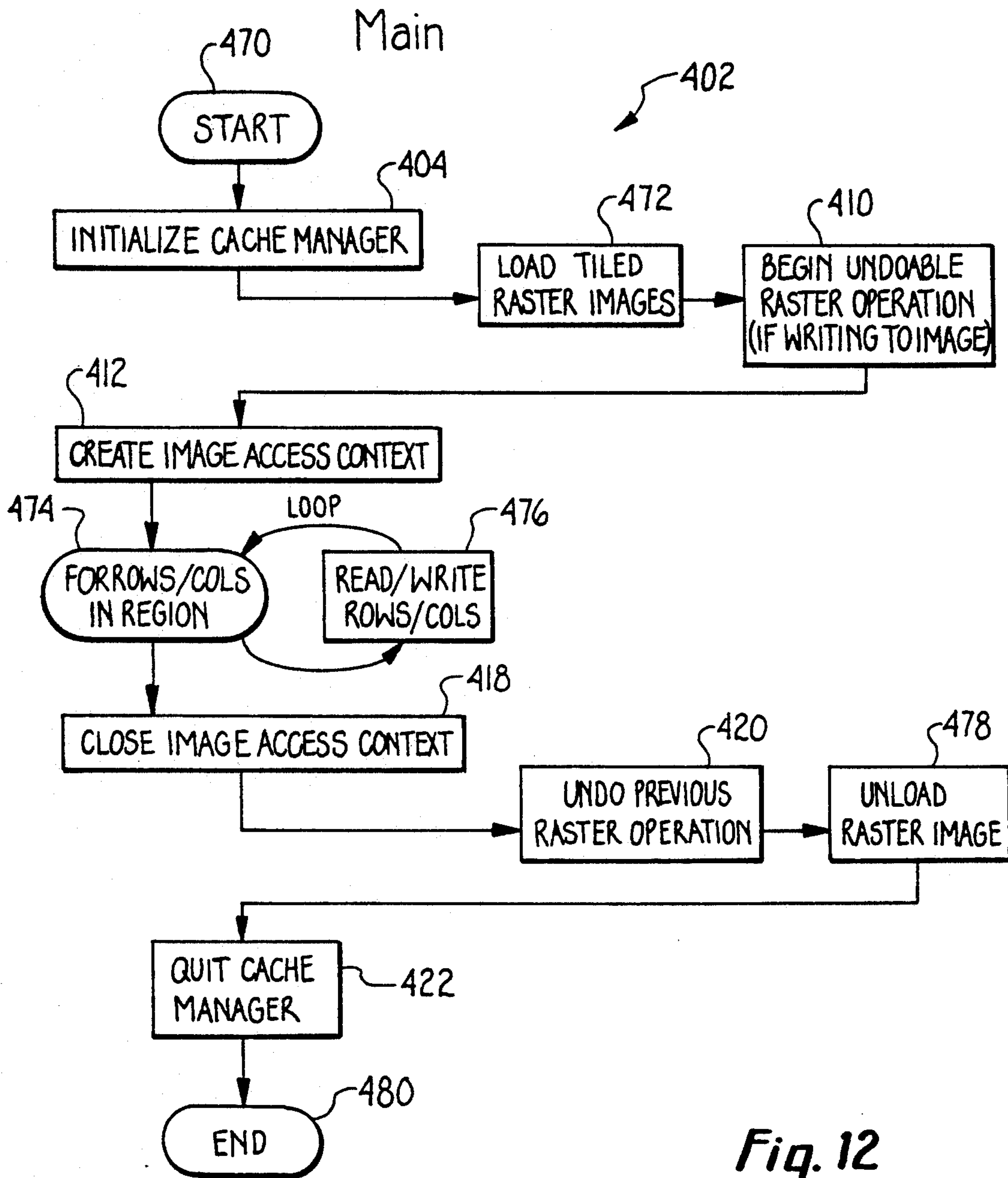
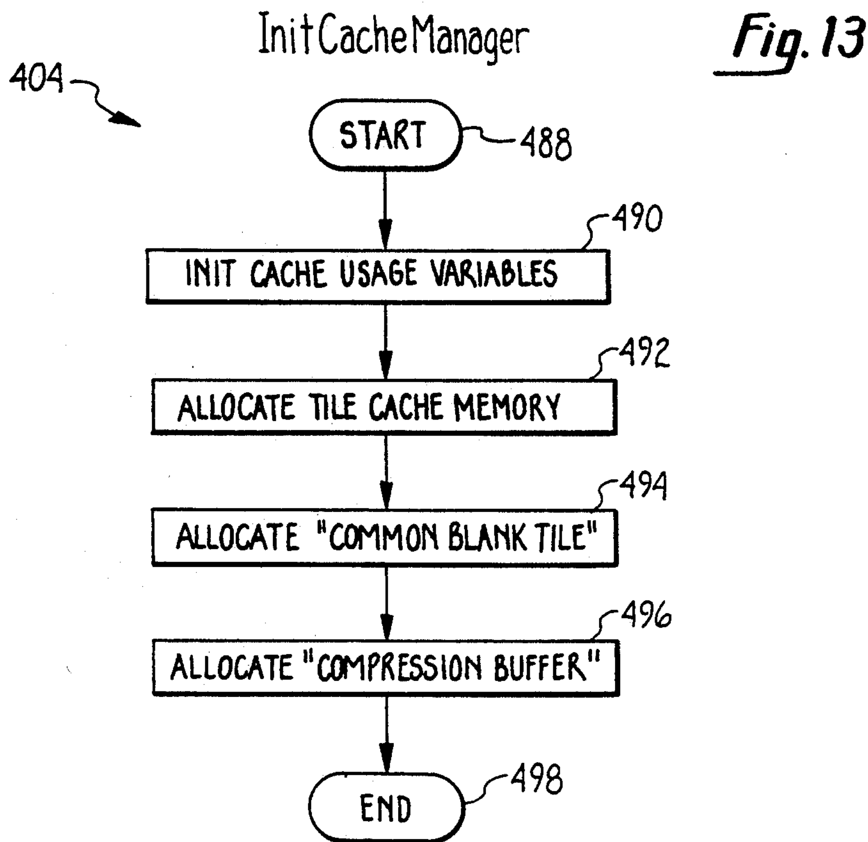
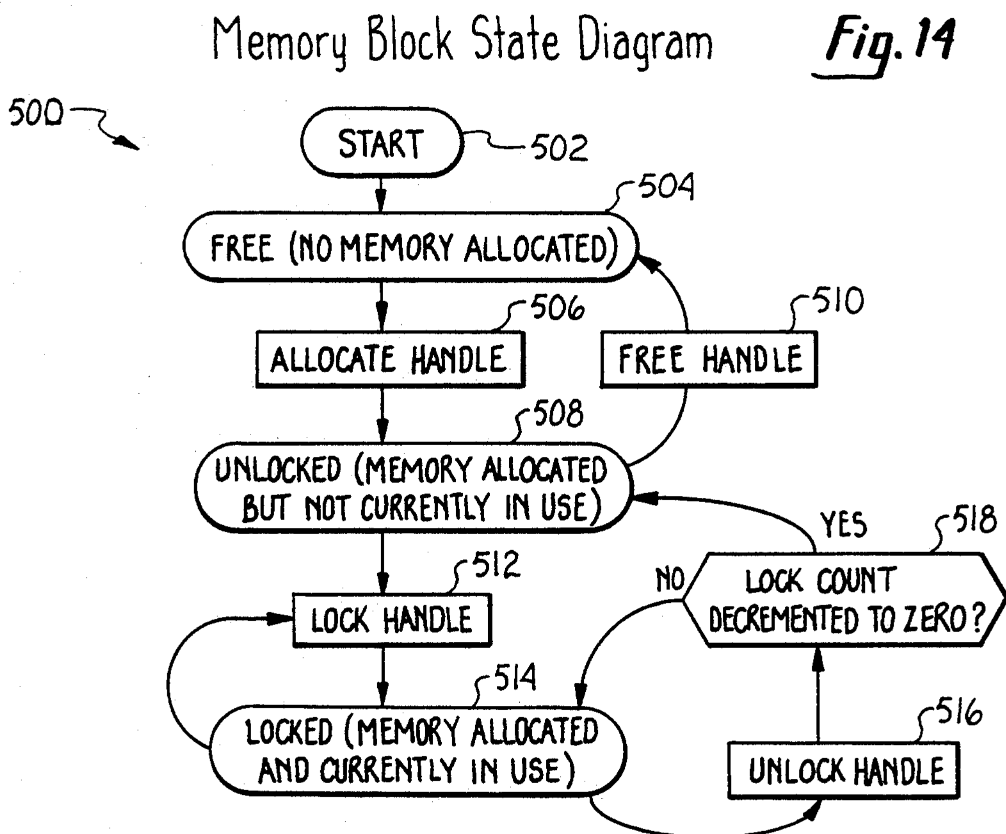
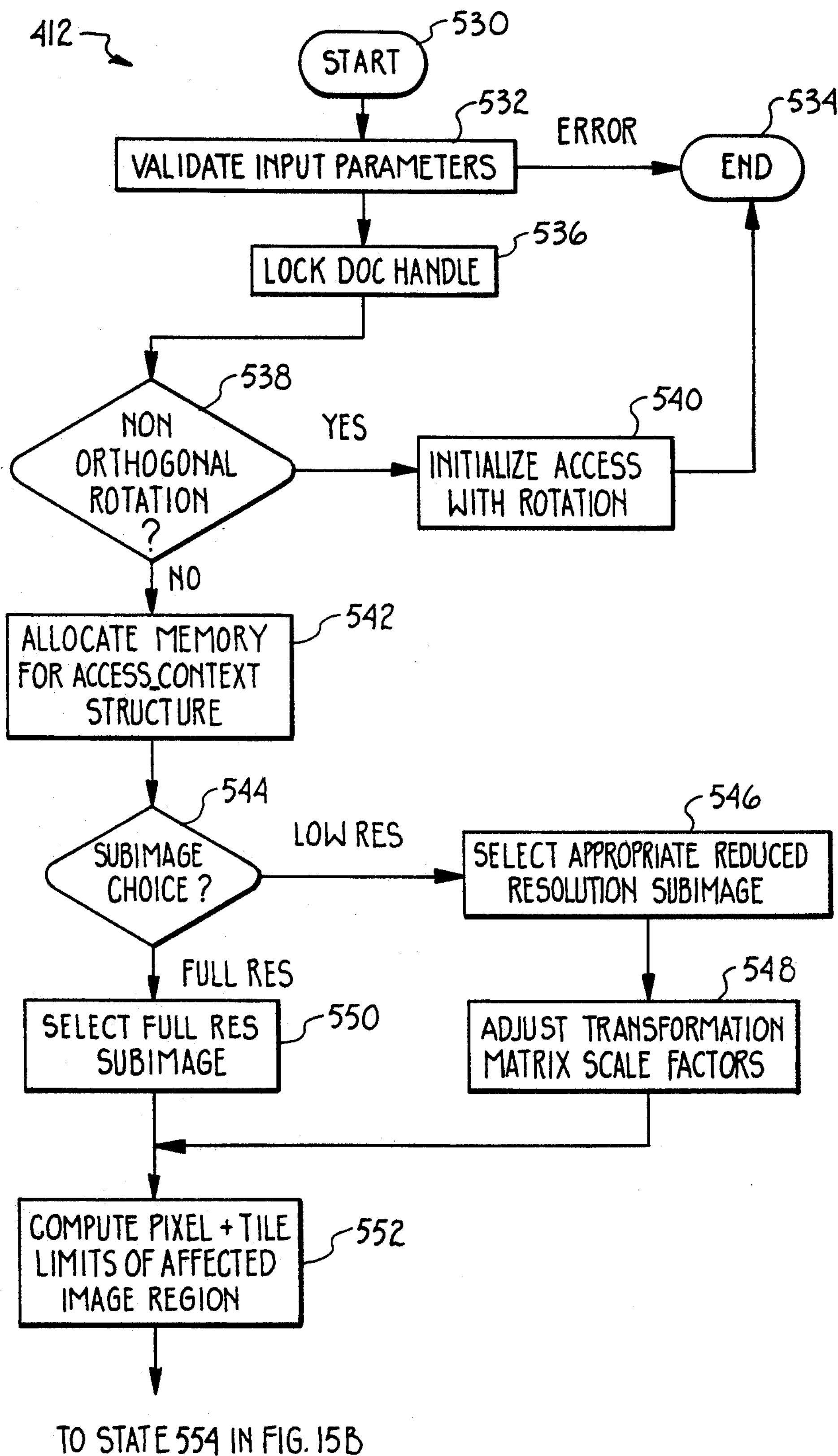
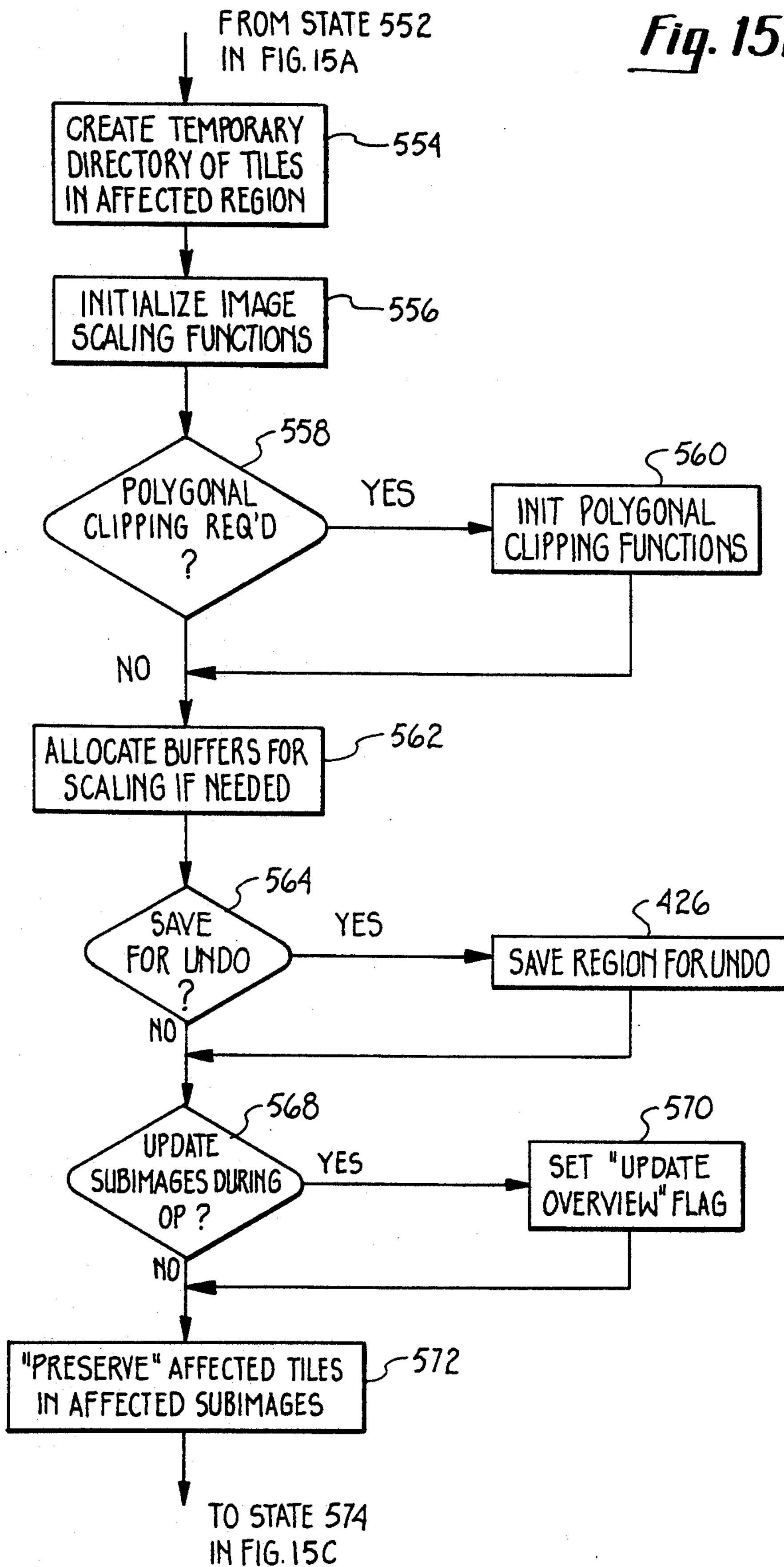


Fig. 12


InitCacheManager
Fig. 13
404
START
488
490
INIT CACHE USAGE VARIABLES
492
ALLOCATE TILE CACHE MEMORY
494
ALLOCATE "COMMON BLANK TILE"
496
ALLOCATE "COMPRESSION BUFFER"
498
END


Memory Block State Diagram
Fig. 14
500
START
502
504
FREE (NO MEMORY ALLOCATED)
506
ALLOCATE HANDLE
510
FREE HANDLE
508
UNLOCKED (MEMORY ALLOCATED BUT NOT CURRENTLY IN USE)
YES
518
NO
LOCK COUNT DECREMENTED TO ZERO?
512
LOCK HANDLE
514
LOCKED (MEMORY ALLOCATED AND CURRENTLY IN USE)
516
UNLOCK HANDLE


Init Image Access
Fig. 15A
412
START
530
VALIDATE INPUT PARAMETERS
532
ERROR
END
534
LOCK DOC HANDLE
536
NON ORTHOGONAL ROTATION ?
538
YES
INITIALIZE ACCESS WITH ROTATION
540
NO
ALLOCATE MEMORY FOR ACCESS_CONTEXT STRUCTURE
542
SUBIMAGE CHOICE ?
544
LOW RES
SELECT APPROPRIATE REDUCED RESOLUTION SUBIMAGE
546
FULL RES
SELECT FULL RES SUBIMAGE
550
ADJUST TRANSFORMATION MATRIX SCALE FACTORS
548
COMPUTE PIXEL + TILE LIMITS OF AFFECTED IMAGE REGION
552
TO STATE 554 IN FIG. 15B


FROM STATE 552
IN FIG. 15A
Fig. 15B
CREATE TEMPORARY DIRECTORY OF TILES IN AFFECTED REGION
554
INITIALIZE IMAGE SCALING FUNCTIONS
556
558
POLYGONAL CLIPPING REQ'D ?
YES
560
INIT POLYGONAL CLIPPING FUNCTIONS
NO
ALLOCATE BUFFERS FOR SCALING IF NEEDED
562
564
SAVE FOR UNDO ?
YES
426
SAVE REGION FOR UNDO
NO
568
UPDATE SUBIMAGES DURING OP ?
YES
570
SET "UPDATE OVERVIEW" FLAG
NO
"PRESERVE" AFFECTED TILES IN AFFECTED SUBIMAGES
572
TO STATE 574
IN FIG. 15C

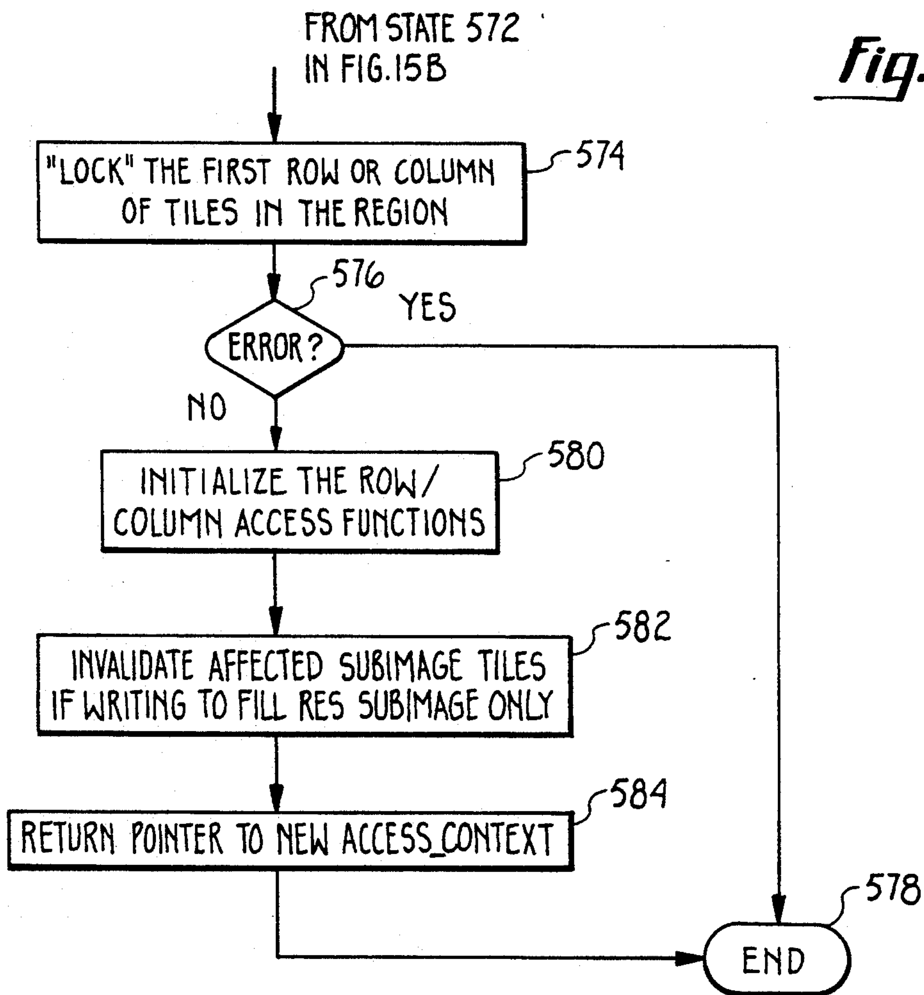


Fig. 15C

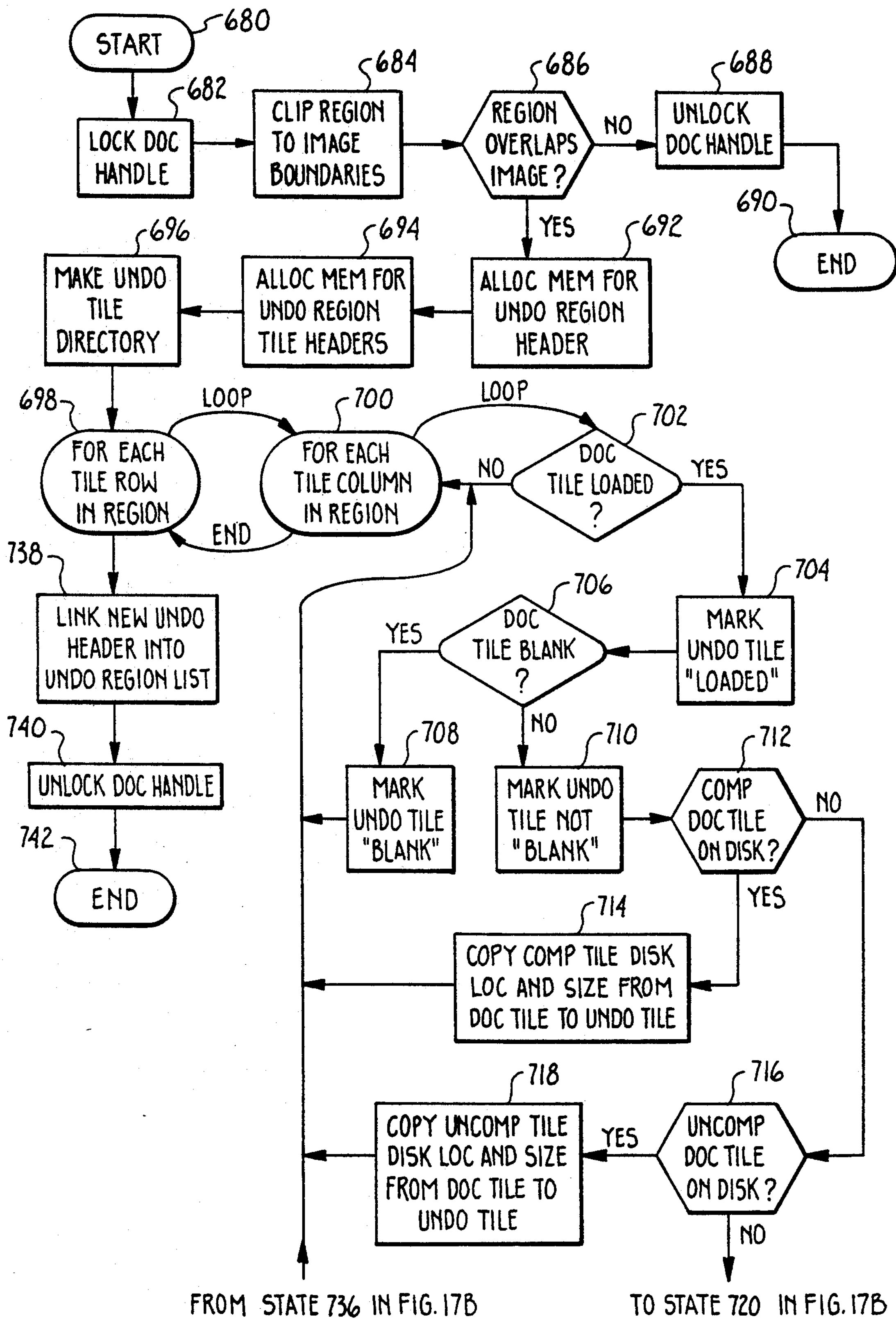
 Unified Patents Exhibit 1005 App'x A-N

Access Context

600

Fig. 16

POINTER TO AFFECTED DOC 602
"SUBIMAGE CHOICE" OPTION VALUE 604
INDEX OF AFFECTED SUBIMAGE 606
ACCESS QUANTUM 608
READ/WRITE OPTION 610
BASIC ORTHOGONAL ROTATION VALUE 612
PIXEL COMBINATION OPERATION 614
SCALER TYPE OPERATION 616
"UPDATE OVERVIEWS" FLAG 618
I/O BUFFER WIDTH & HEIGHT 620
I/O BUFFER PITCH (BYTES/ROW) 622
I/O BUFFER BIT OFFSET TO START OF RUN 624
ROWS PER STRIP (FOR "AQ_STRIP" ACCESS QUANTUM) 626
NUMBER OF I/O BUFFER ROWS YET TO BE PROCESSED 628
POINTER TO ACCESS FUNCTION USED IN "SeqBufImageAccess" 630
STEPPING DIRECTIONS FOR IMAGE ROW AND COLUMN INDICIES 632
POINTER TO POLYGON CLIPPING INFORMATION 634
POINTER TO RASTER SCALING INFORMATION 636
POINTER TO UNCOMPRESSED DATA IN CURRENTLY LOCKED TILES 638
POINTER TO REGION TILE DIRECTORY 640
NEXT IMAGE ROW & COLUMN TO BE ACCESSED 642
TERMINAL ROW & COLUMN OF ACCESS REGION 644
UNCLIPPED EXTENT OF ACCESS REGION 646
CLIPPED EXTENT OF ACCESS REGION 648
CLIPPED IMAGE BUFFER BIT OFFSET AND LENGTH 650
NUMBER OF TILE ROWS & COLS IN ACCESS REGION 652
ROW & COLUMN OF CURRENTLY LOCKED TILES 654
IMAGE ROW & COL AT ORIGIN OF FIRST TILE IN ACCESS REGION 656
NUMBER OF I/O BUFFER ROWS HELD OVER FOR NEXT STRIP 658
POINTER TO IMAGE TILING/UNTILING BUFFER 660
NUMBER OF BYTES IN TILING/UNTILING BUFFER 662
BIT OFFSET FOR TILING/UNTILING BUFFER 664
ACCESS TRANSFORMATION MATRIX 666



 Unified Patents Exhibit 1005 App'x A-N

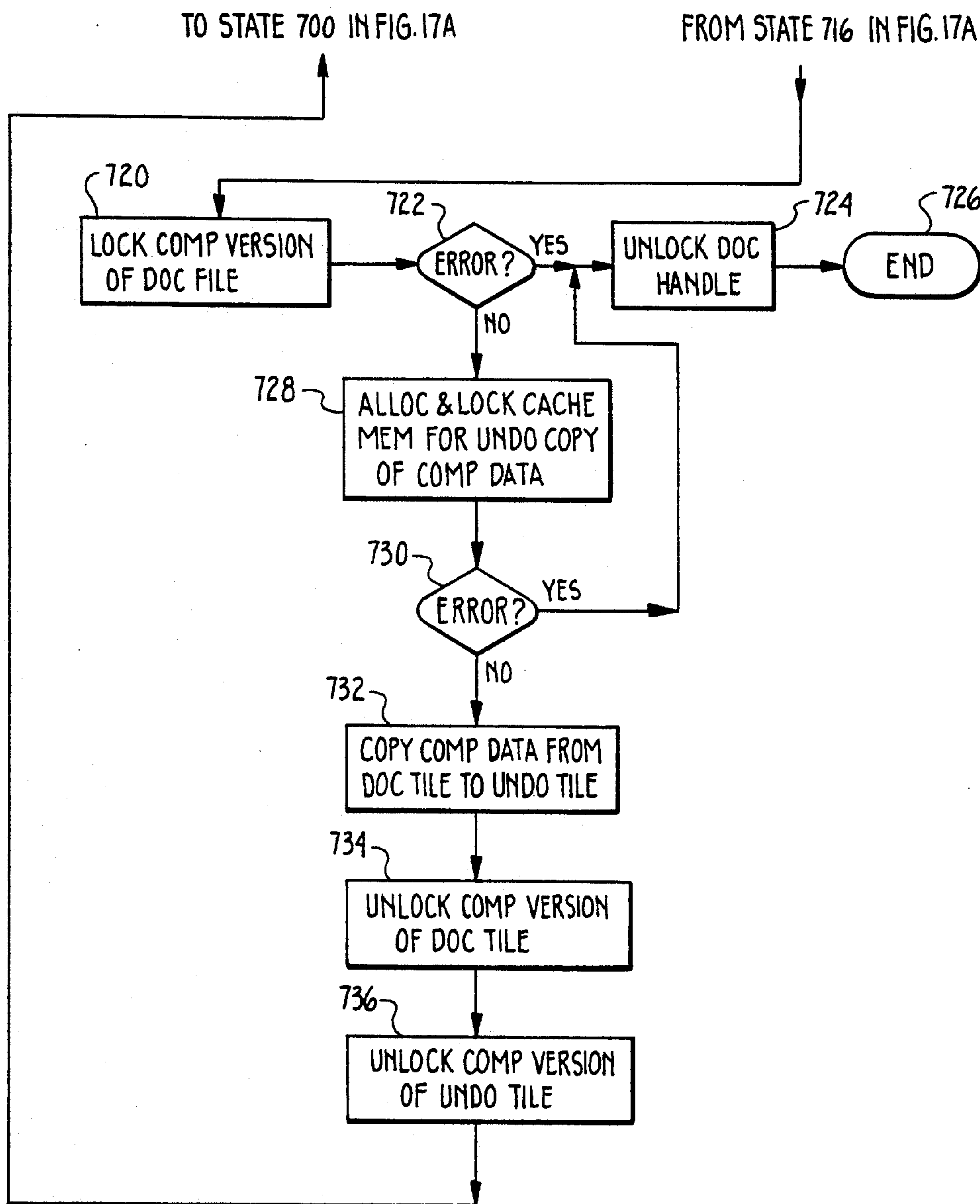
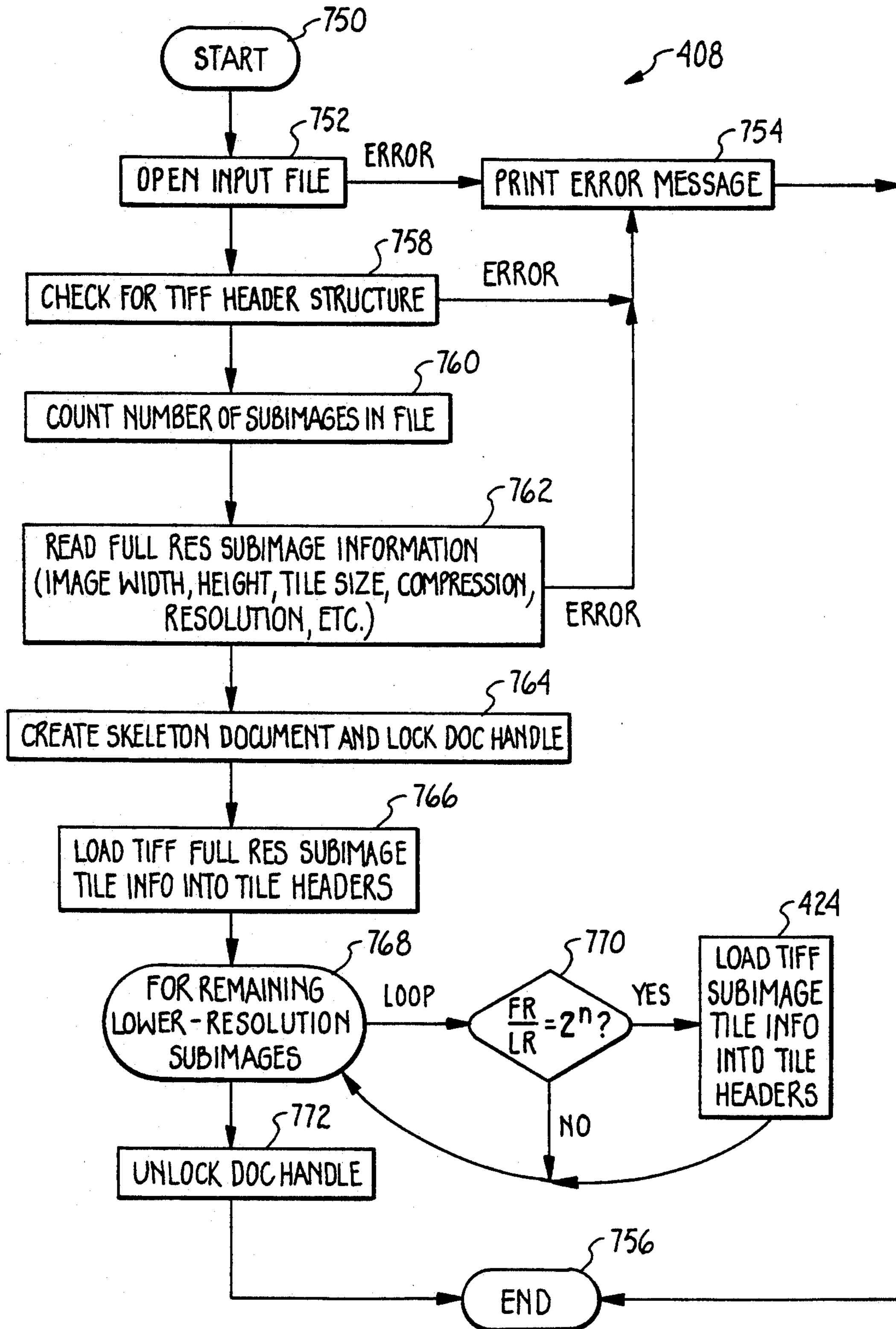
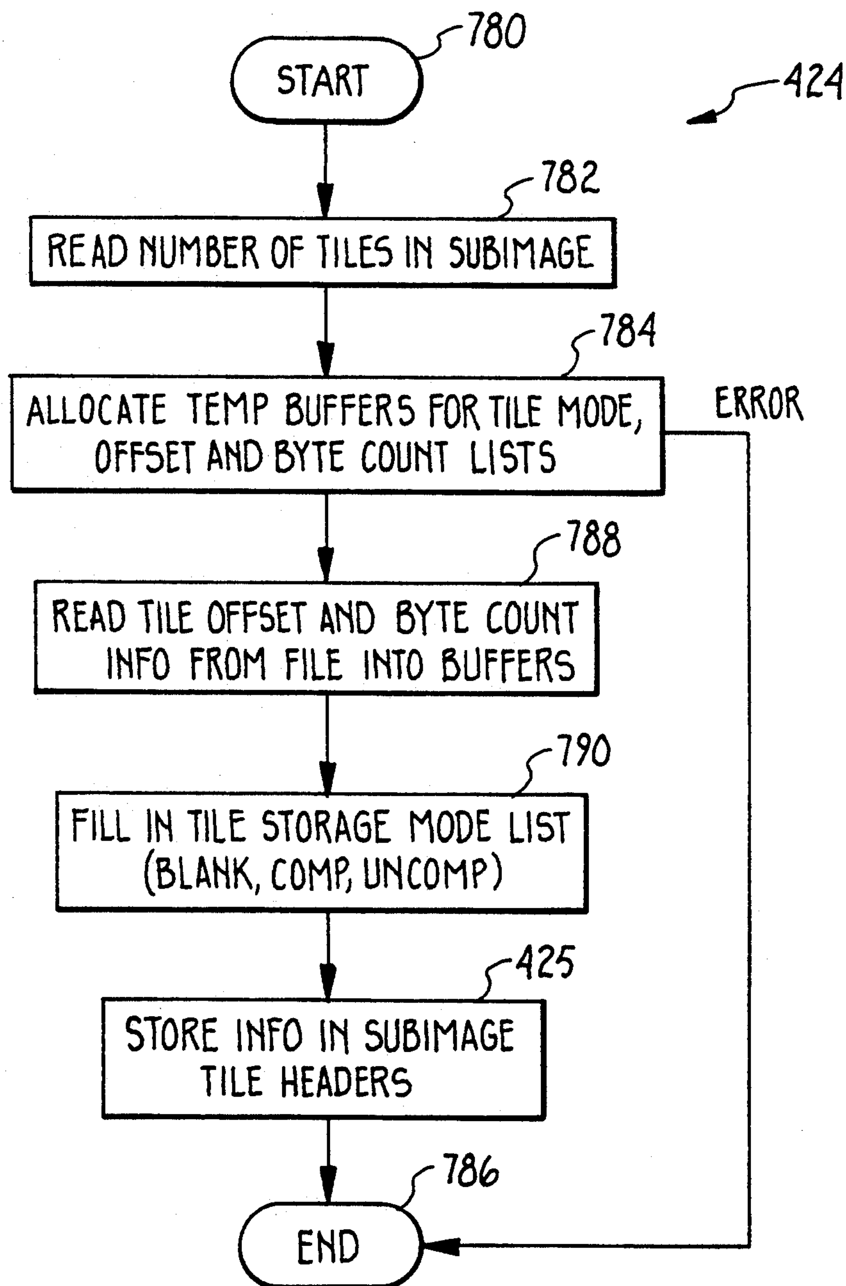
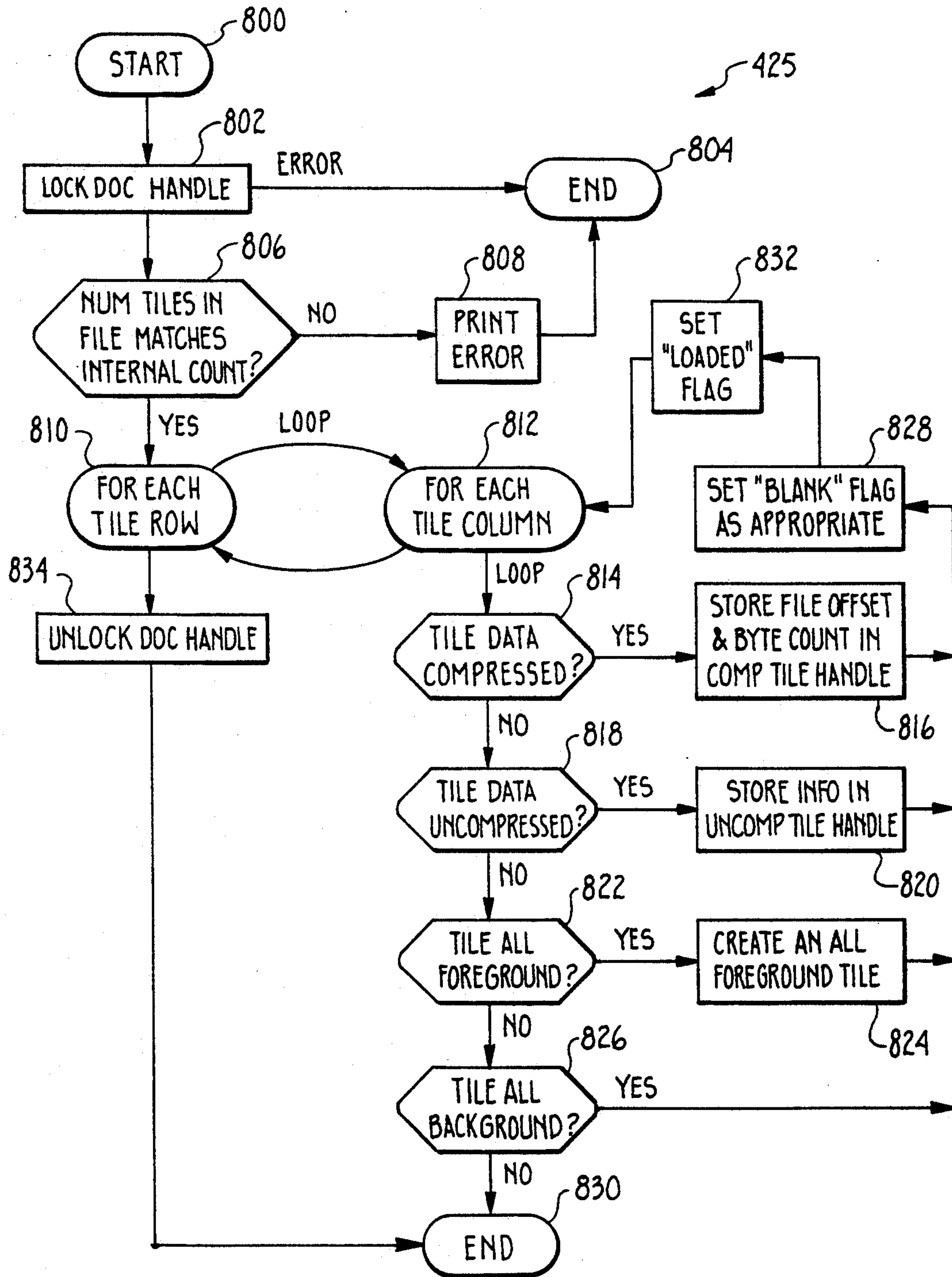


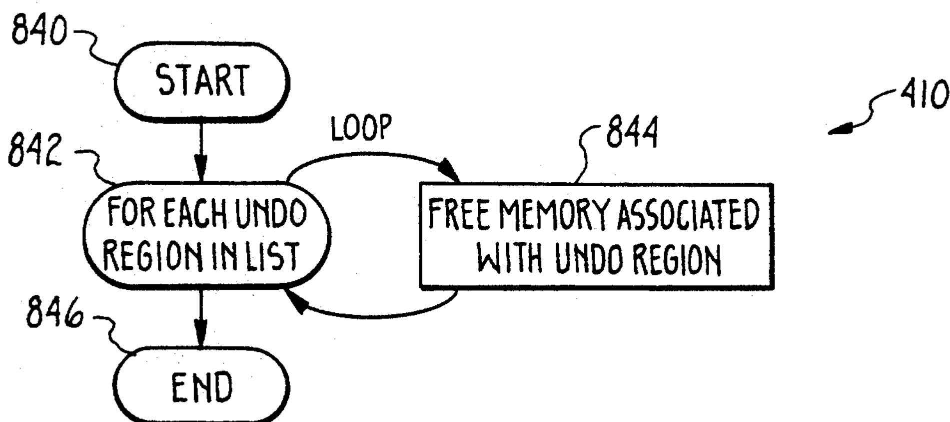
Fig. 17B



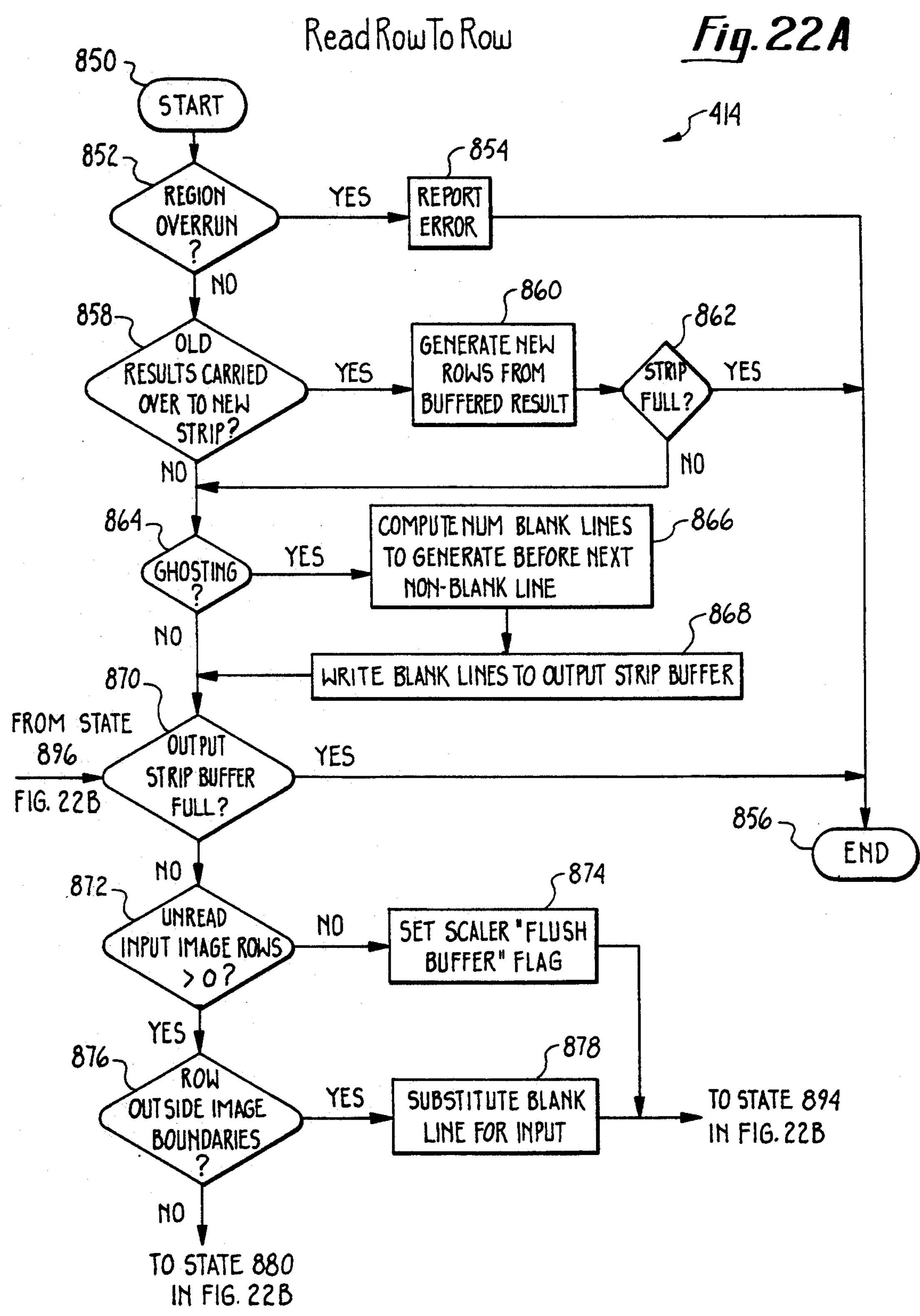
 Unified Patents Exhibit 1005 App'x A-N


LoadTiff Tiles Std
Fig. 19
780
START
424
782
READ NUMBER OF TILES IN SUBIMAGE
784
ALLOCATE TEMP BUFFERS FOR TILE MODE, OFFSET AND BYTE COUNT LISTS
ERROR
788
READ TILE OFFSET AND BYTE COUNT INFO FROM FILE INTO BUFFERS
790
FILL IN TILE STORAGE MODE LIST (BLANK, COMP, UNCOMP)
425
STORE INFO IN SUBIMAGE TILE HEADERS
786
END


LoadSubImDiskCache
Fig. 20
800
START
425
802
LOCK DOC HANDLE
ERROR
END
804
806
NUM TILES IN FILE MATCHES INTERNAL COUNT?
NO
808
PRINT ERROR
832
SET "LOADED" FLAG
810
YES
LOOP
812
FOR EACH TILE ROW
FOR EACH TILE COLUMN
828
SET "BLANK" FLAG AS APPROPRIATE
834
UNLOCK DOC HANDLE
LOOP
814
TILE DATA COMPRESSED?
YES
STORE FILE OFFSET & BYTE COUNT IN COMP TILE HANDLE
816
NO
818
TILE DATA UNCOMPRESSED?
YES
STORE INFO IN UNCOMP TILE HANDLE
820
NO
822
TILE ALL FOREGROUND?
YES
CREATE AN ALL FOREGROUND TILE
824
NO
826
TILE ALL BACKGROUND?
YES
NO
830
END



BeginUndoableRasOp
Fig. 21
840
START
842
FOR EACH UNDO REGION IN LIST
LOOP
844
FREE MEMORY ASSOCIATED WITH UNDO REGION
410
846
END


Read Row To Row
Fig. 22A
414
850
START
852
REGION OVERRUN ?
YES
854
REPORT ERROR
NO
858
OLD RESULTS CARRIED OVER TO NEW STRIP?
YES
860
GENERATE NEW ROWS FROM BUFFERED RESULT
862
STRIP FULL?
YES
NO
NO
864
GHOSTING ?
YES
866
COMPUTE NUM BLANK LINES TO GENERATE BEFORE NEXT NON-BLANK LINE
NO
868
WRITE BLANK LINES TO OUTPUT STRIP BUFFER
870
FROM STATE 896 FIG. 22B
OUTPUT STRIP BUFFER FULL?
YES
856
END
NO
872
UNREAD INPUT IMAGE ROWS > 0?
NO
874
SET SCALER "FLUSH BUFFER" FLAG
YES
876
ROW OUTSIDE IMAGE BOUNDARIES ?
YES
878
SUBSTITUTE BLANK LINE FOR INPUT
TO STATE 894 IN FIG. 22B
NO
TO STATE 880 IN FIG. 22B

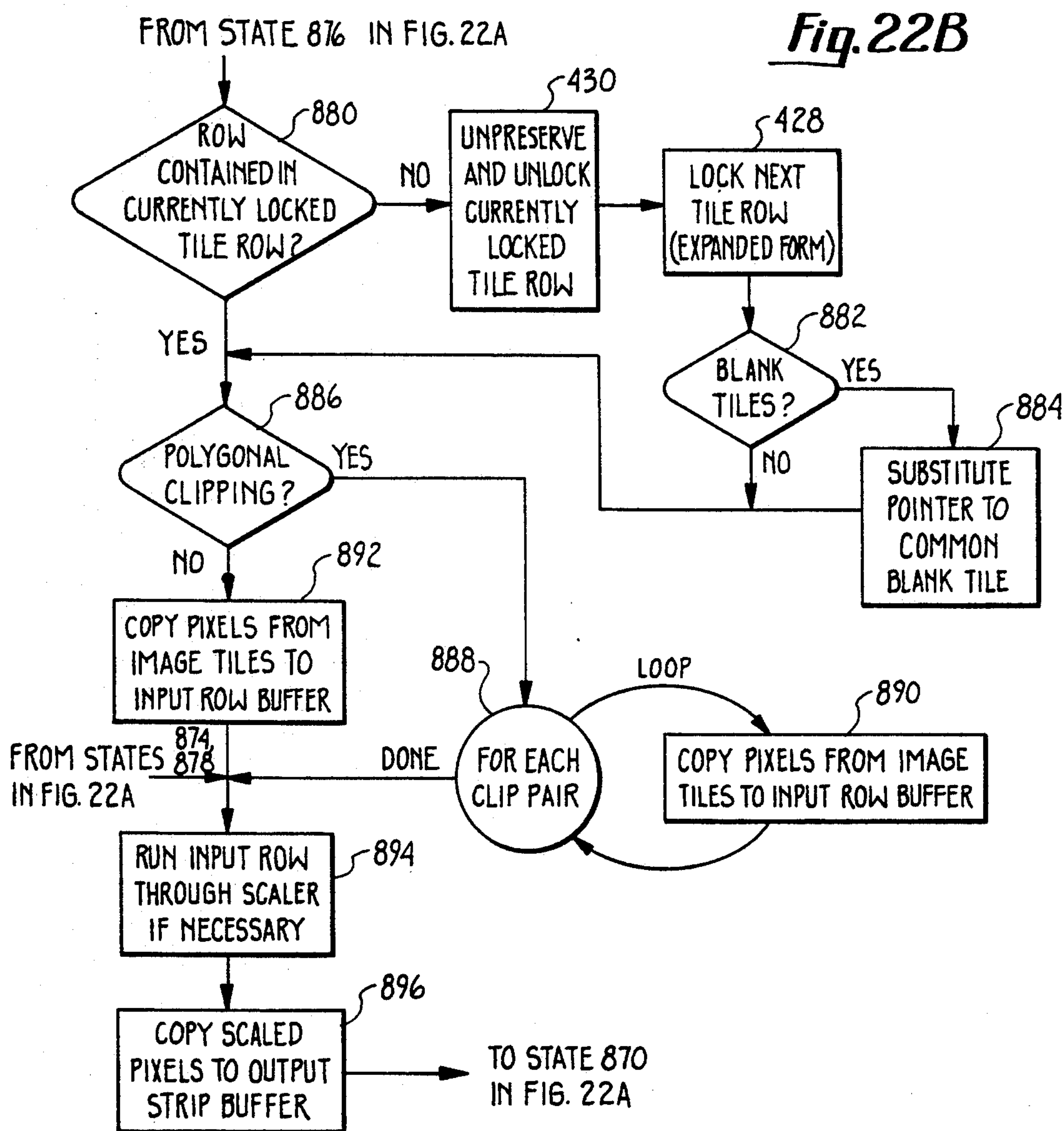
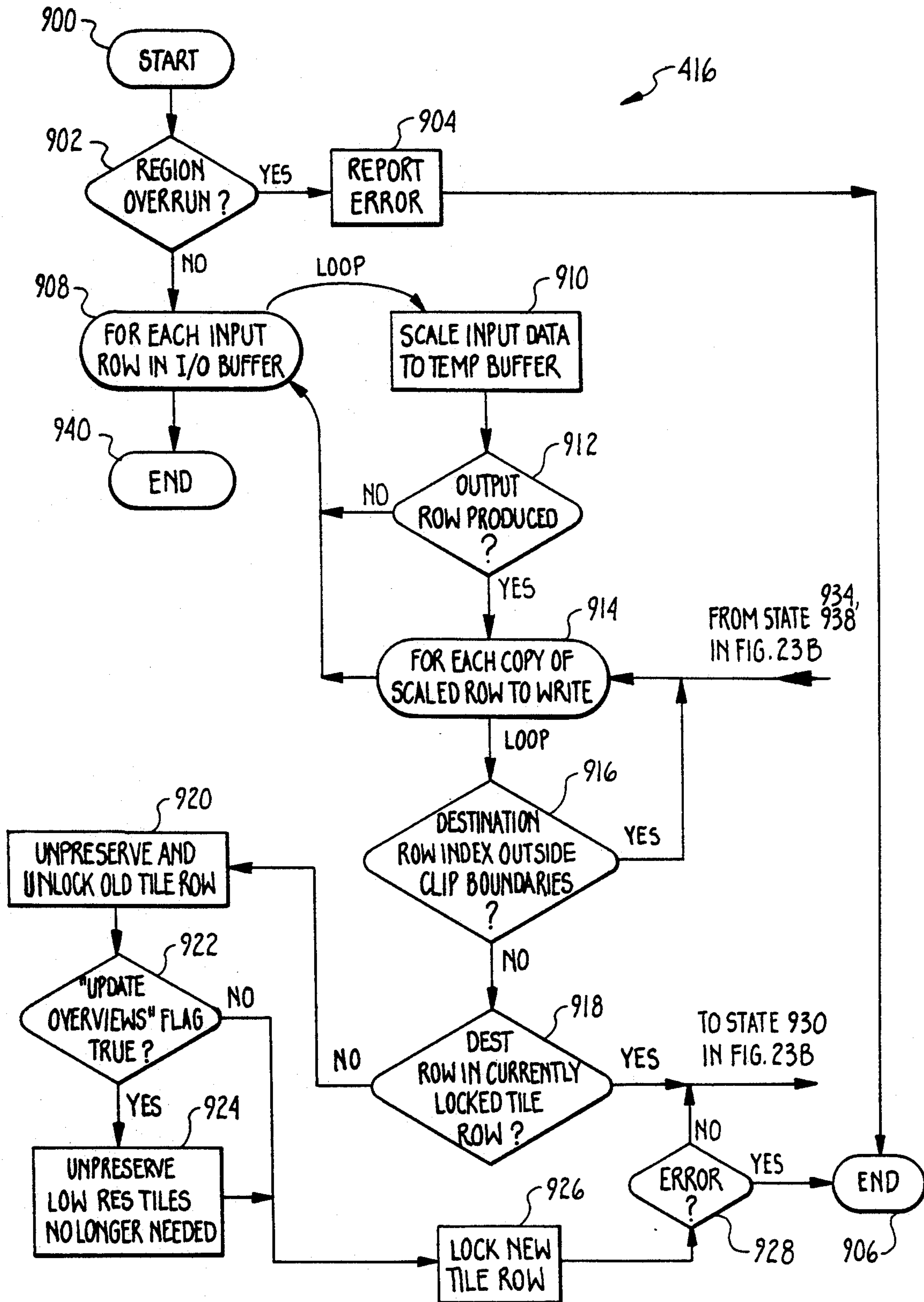
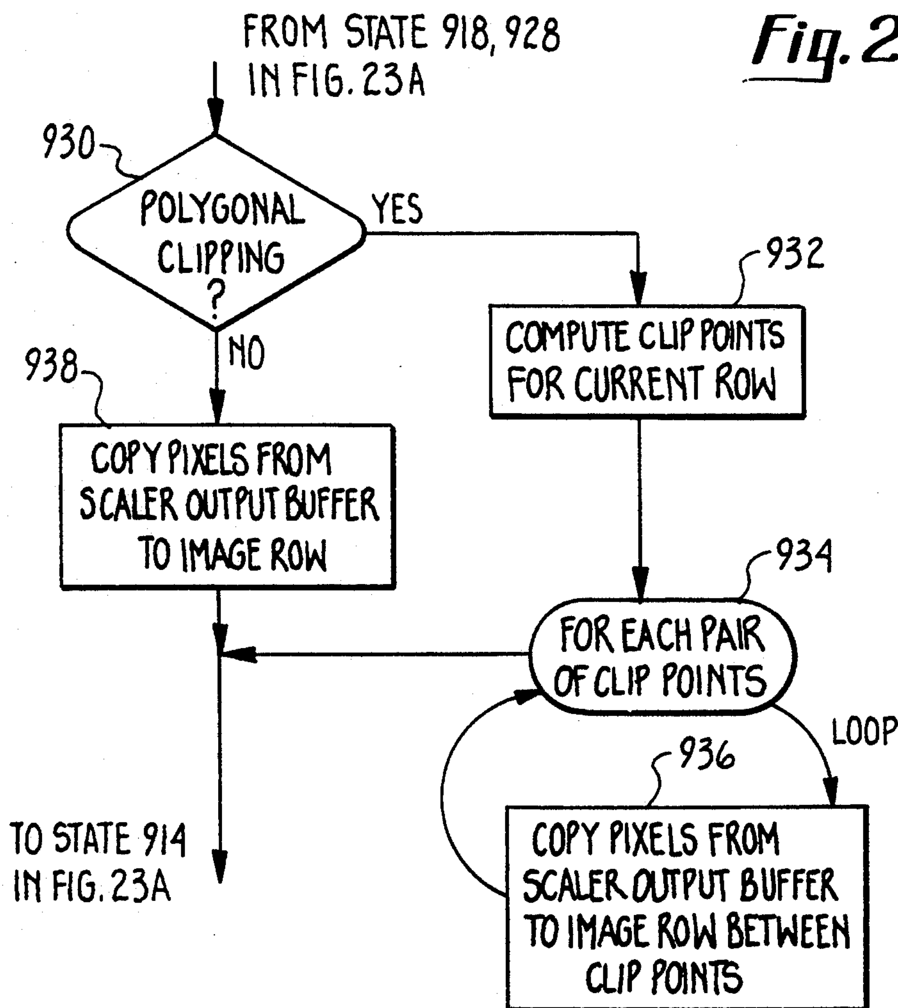
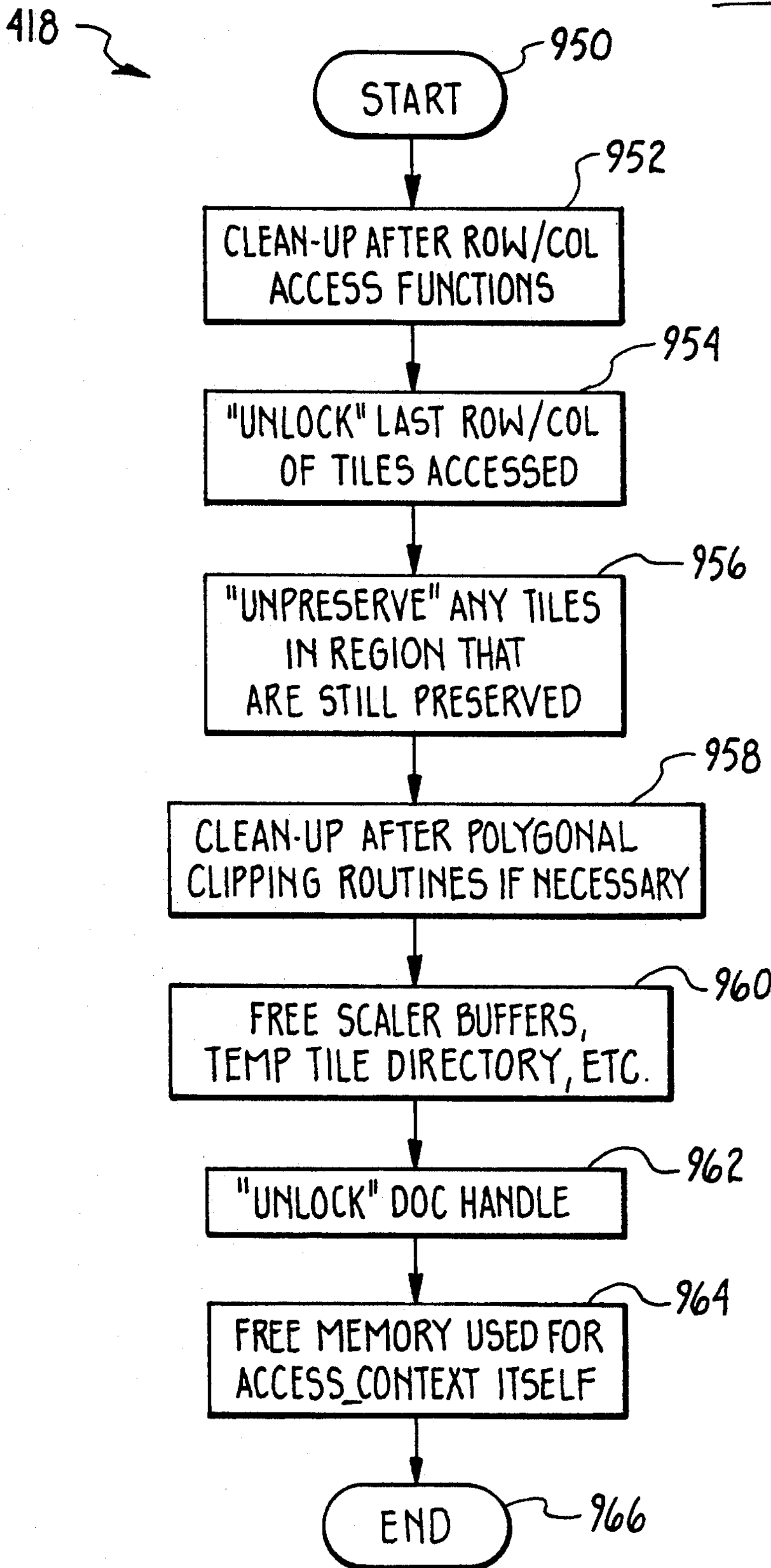


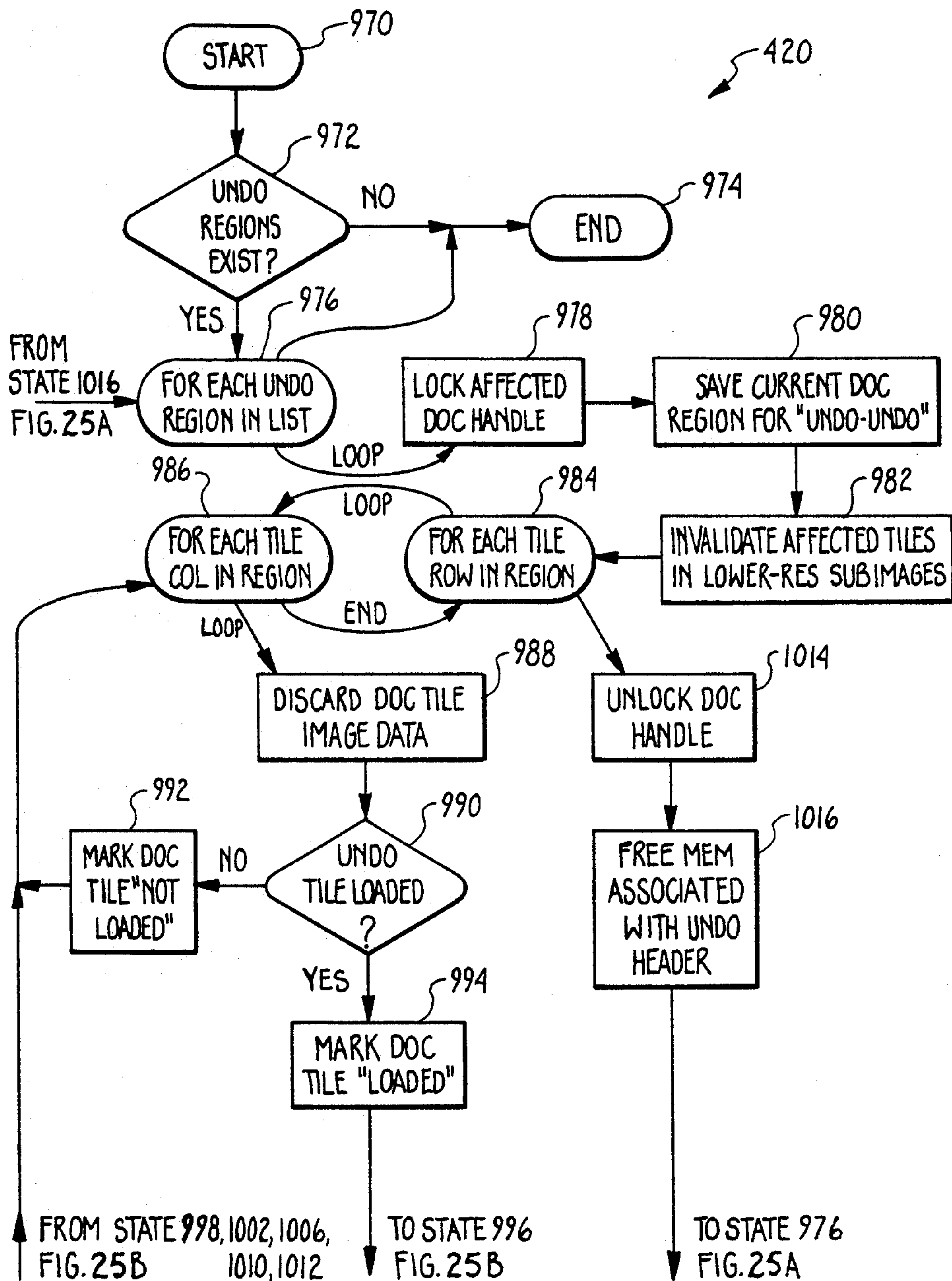
Fig. 22B
FROM STATE 876 IN FIG. 22A
880
ROW CONTAINED IN CURRENTLY LOCKED TILE ROW?
NO
430
UNPRESERVE AND UNLOCK CURRENTLY LOCKED TILE ROW
428
LOCK NEXT TILE ROW (EXPANDED FORM)
882
BLANK TILES?
YES
884
SUBSTITUTE POINTER TO COMMON BLANK TILE
NO
YES
886
POLYGONAL CLIPPING?
YES
NO
892
COPY PIXELS FROM IMAGE TILES TO INPUT ROW BUFFER
888
LOOP
890
FOR EACH CLIP PAIR
COPY PIXELS FROM IMAGE TILES TO INPUT ROW BUFFER
DONE
FROM STATES 874 878 IN FIG. 22A
RUN INPUT ROW THROUGH SCALER IF NECESSARY
894
896
COPY SCALED PIXELS TO OUTPUT STRIP BUFFER
TO STATE 870 IN FIG. 22A



 Unified Patents Exhibit 1005 App'x A-N


FROM STATE 918, 928
IN FIG. 23A
Fig. 23B
930
POLYGONAL CLIPPING ?
YES
NO
932
COMPUTE CLIP POINTS FOR CURRENT ROW
938
COPY PIXELS FROM SCALER OUTPUT BUFFER TO IMAGE ROW
934
FOR EACH PAIR OF CLIP POINTS
LOOP
936
COPY PIXELS FROM SCALER OUTPUT BUFFER TO IMAGE ROW BETWEEN CLIP POINTS
TO STATE 914
IN FIG. 23A


End ImageAccess
Fig.24
418
START
950
CLEAN-UP AFTER ROW/COL ACCESS FUNCTIONS
952
"UNLOCK" LAST ROW/COL OF TILES ACCESSED
954
"UNPRESERVE" ANY TILES IN REGION THAT ARE STILL PRESERVED
956
CLEAN-UP AFTER POLYGONAL CLIPPING ROUTINES IF NECESSARY
958
FREE SCALER BUFFERS, TEMP TILE DIRECTORY, ETC.
960
"UNLOCK" DOC HANDLE
962
FREE MEMORY USED FOR ACCESS_CONTEXT ITSELF
964
END
966


UndoPreviousRasOp
Fig. 25A
420
START
970
UNDO REGIONS EXIST?
972
NO
END
974
YES
976
FROM STATE 1016 FIG. 25A
FOR EACH UNDO REGION IN LIST
LOOP
LOCK AFFECTED DOC HANDLE
978
SAVE CURRENT DOC REGION FOR "UNDO-UNDO"
980
INVALIDATE AFFECTED TILES IN LOWER-RES SUBIMAGES
982
986
LOOP
984
FOR EACH TILE COL IN REGION
FOR EACH TILE ROW IN REGION
END
LOOP
DISCARD DOC TILE IMAGE DATA
988
UNLOCK DOC HANDLE
1014
992
MARK DOC TILE "NOT LOADED"
NO
UNDO TILE LOADED ?
990
FREE MEM ASSOCIATED WITH UNDO HEADER
1016
YES
994
MARK DOC TILE "LOADED"
FROM STATE 998, 1002, 1006, 1010, 1012 FIG. 25B
TO STATE 996 FIG. 25B
TO STATE 976 FIG. 25A

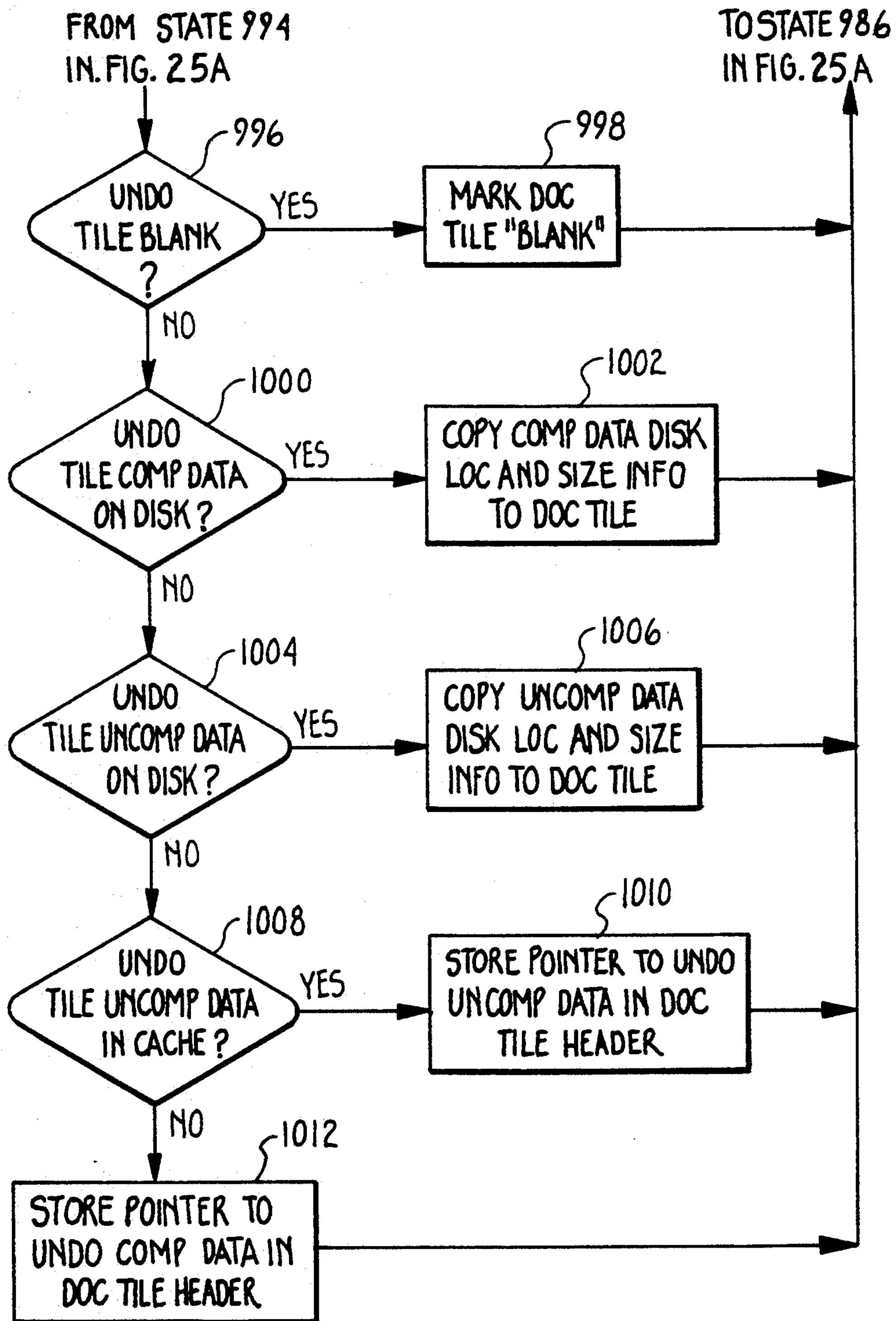
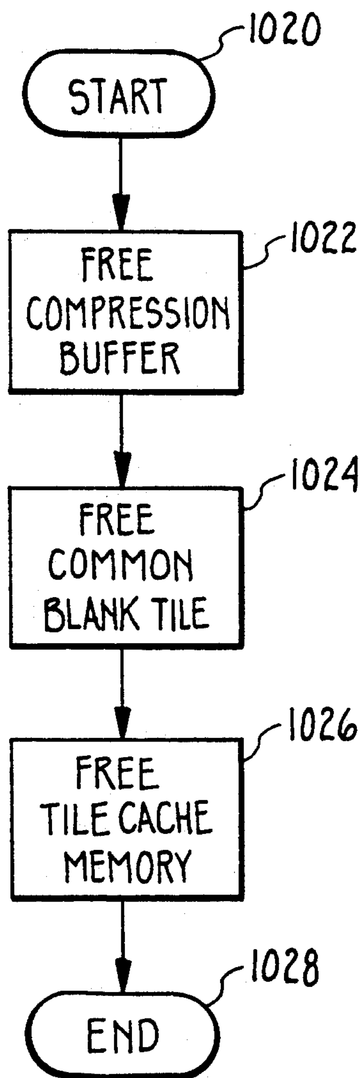
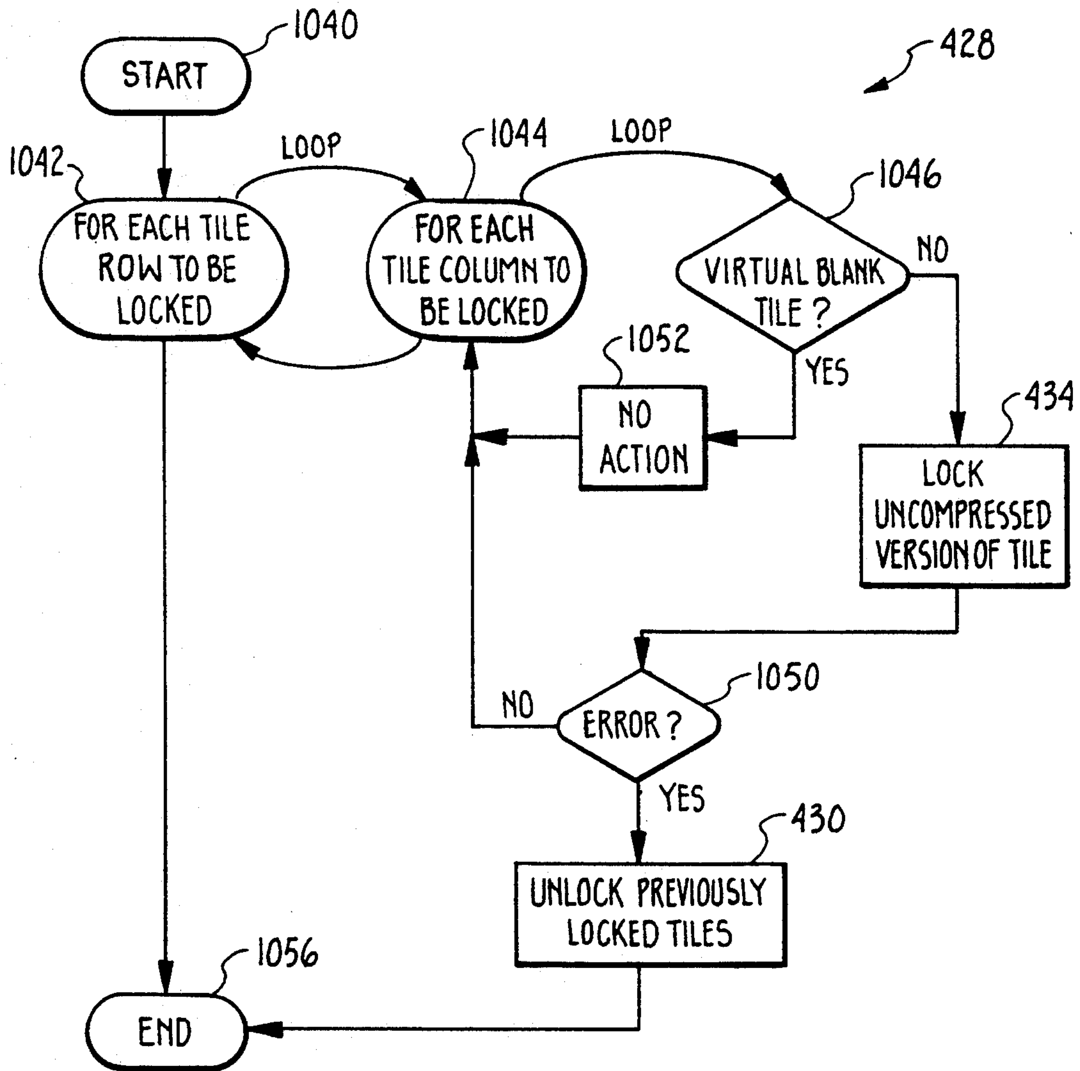


Fig.25B
FROM STATE 994
IN.FIG. 25A
TO STATE 986
IN FIG. 25A
996
UNDO
TILE BLANK
?
YES
998
MARK DOC
TILE "BLANK"
NO
1000
UNDO
TILE COMP DATA
ON DISK ?
YES
1002
COPY COMP DATA DISK
LOC AND SIZE INFO
TO DOC TILE
NO
1004
UNDO
TILE UNCOMP DATA
ON DISK ?
YES
1006
COPY UNCOMP DATA
DISK LOC AND SIZE
INFO TO DOC TILE
NO
1008
UNDO
TILE UNCOMP DATA
IN CACHE ?
YES
1010
STORE POINTER TO UNDO
UNCOMP DATA IN DOC
TILE HEADER
NO
1012
STORE POINTER TO
UNDO COMP DATA IN
DOC TILE HEADER

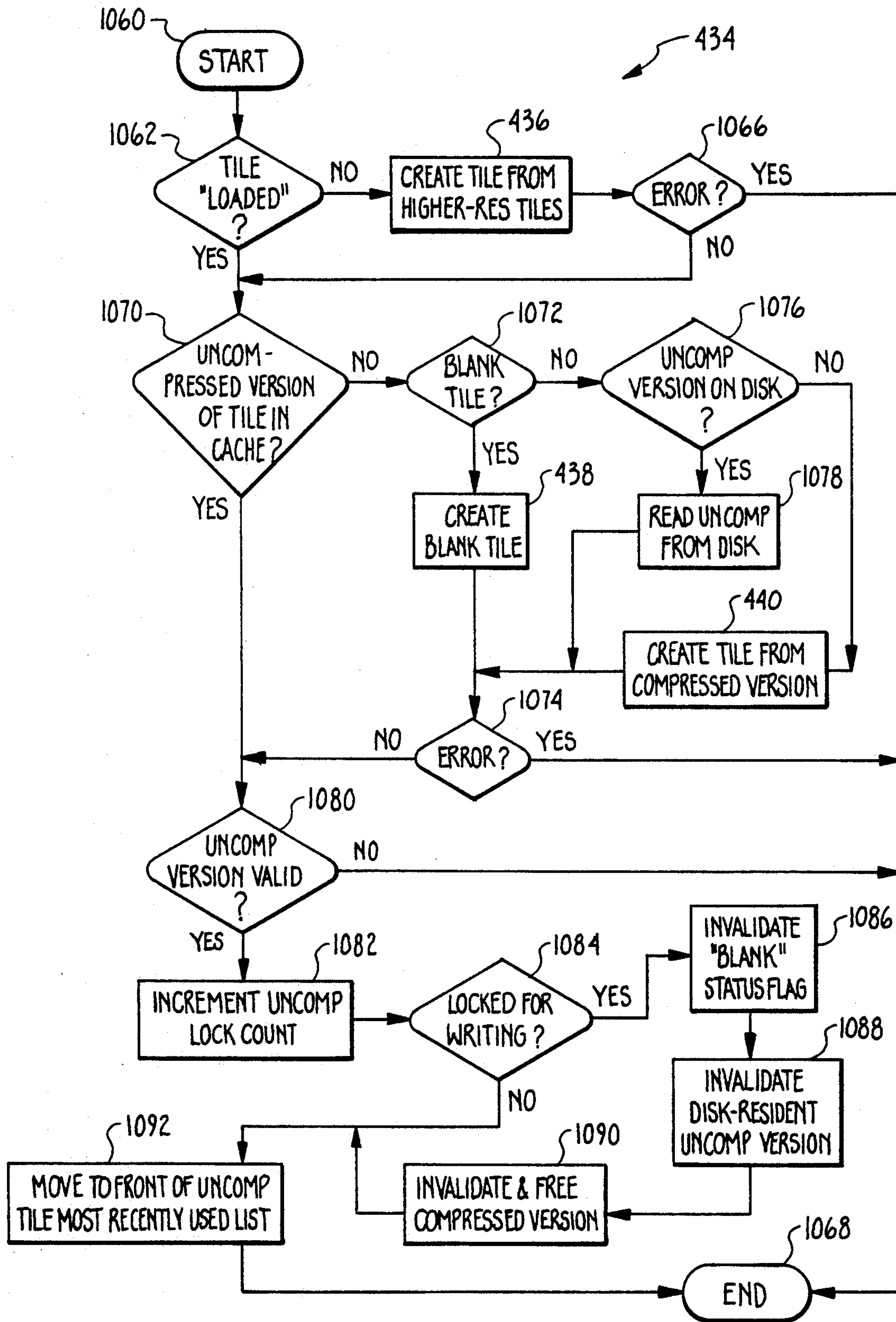
End Cache Manager

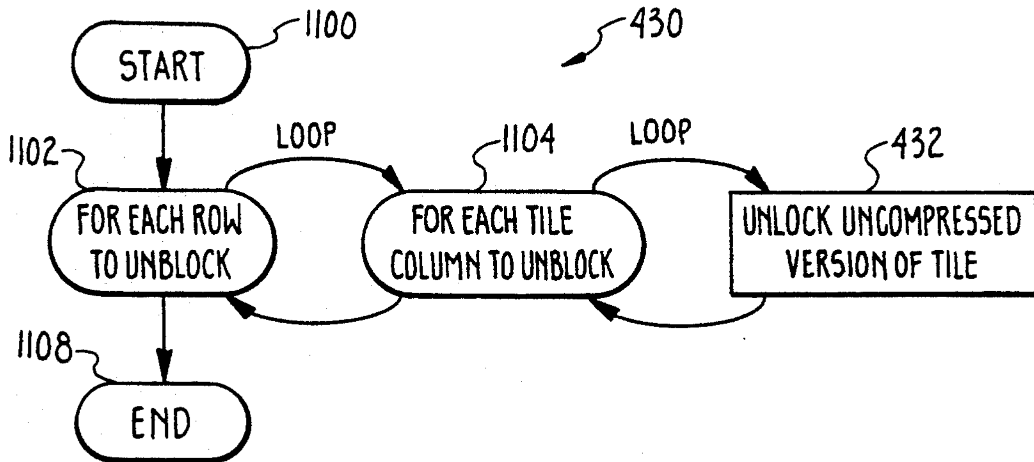
Fig. 26

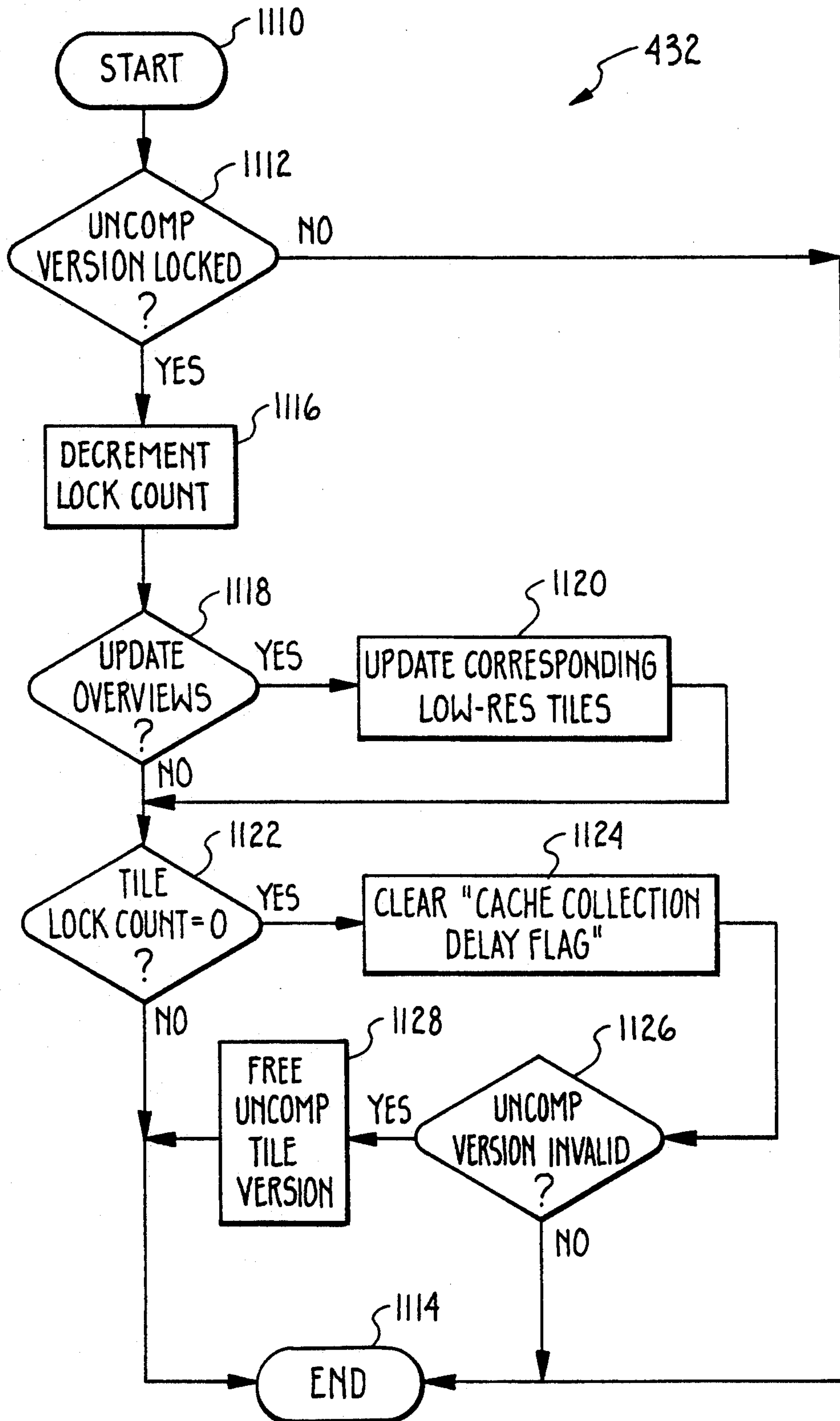


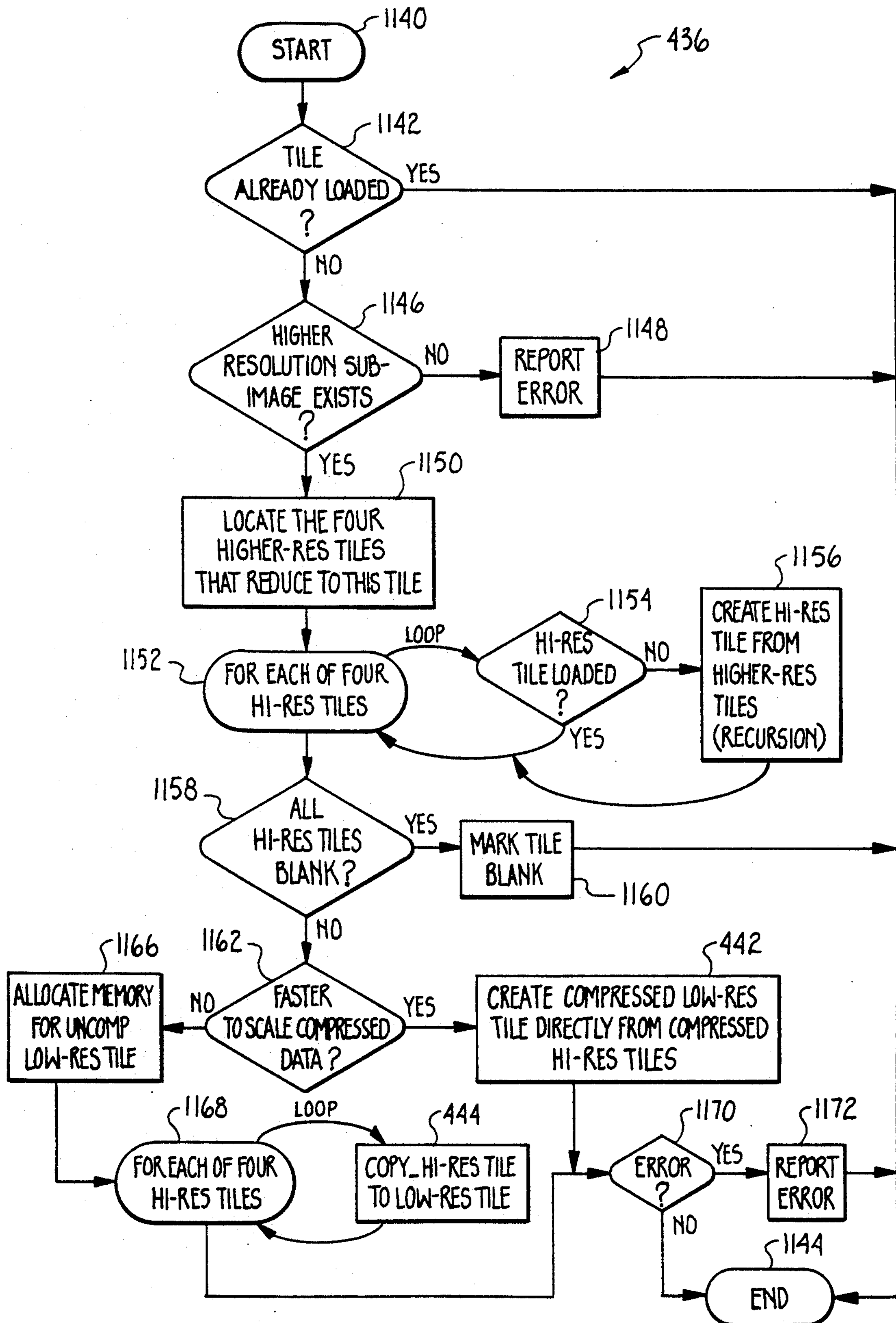


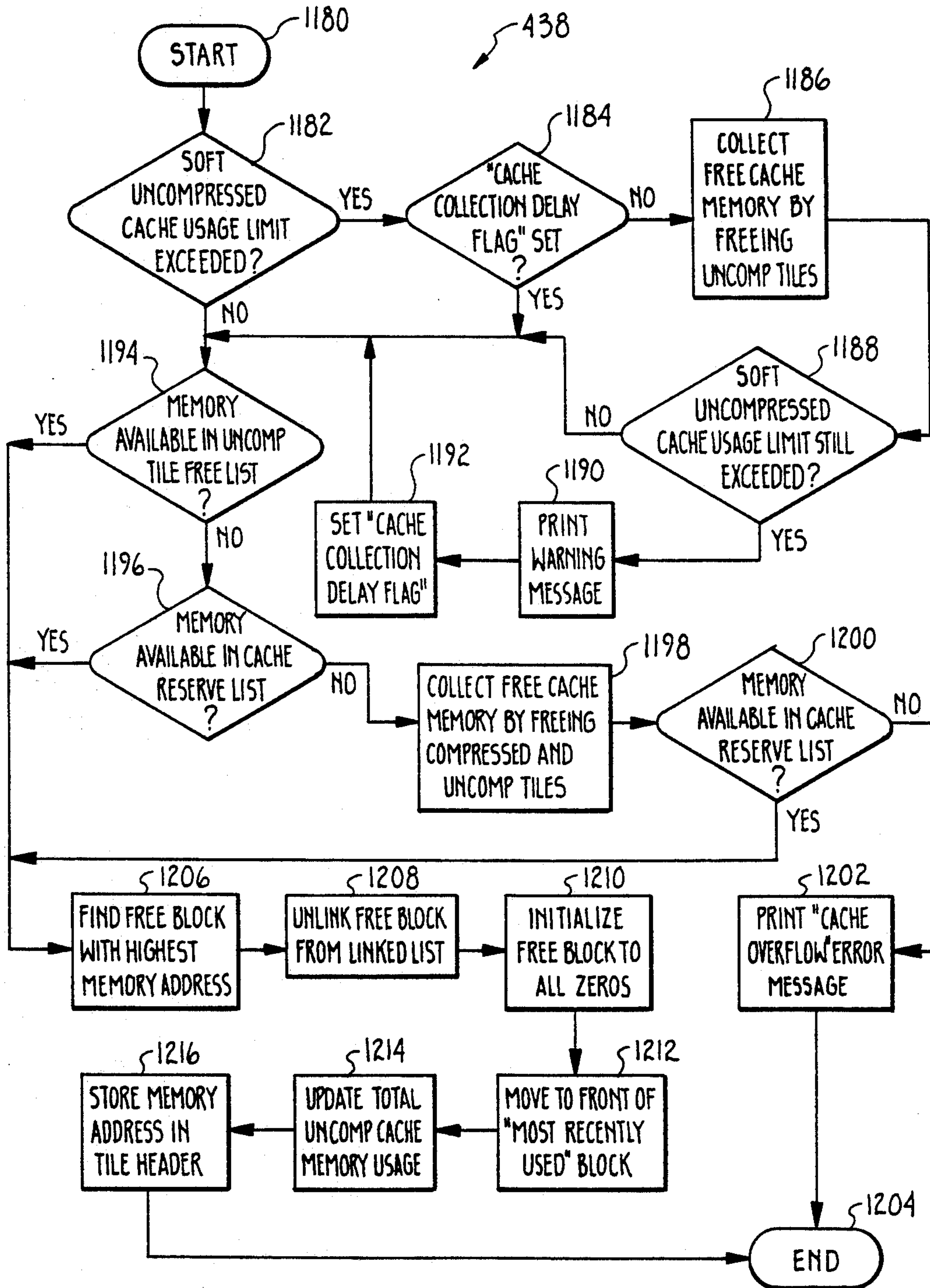
ExpTileLock
Fig. 27
428
1040
START
1042
FOR EACH TILE ROW TO BE LOCKED
LOOP
1044
FOR EACH TILE COLUMN TO BE LOCKED
LOOP
1046
VIRTUAL BLANK TILE ?
NO
YES
1052
NO ACTION
434
LOCK UNCOMPRESSED VERSION OF TILE
1050
ERROR ?
NO
YES
430
UNLOCK PREVIOUSLY LOCKED TILES
1056
END


LockExpHandle
Fig. 28
434
1060
START
1062
TILE "LOADED" ?
NO
YES
436
CREATE TILE FROM HIGHER-RES TILES
1066
ERROR ?
YES
NO
1070
UNCOM-PRESSED VERSION OF TILE IN CACHE ?
NO
YES
1072
BLANK TILE ?
NO
YES
1076
UNCOMP VERSION ON DISK ?
NO
YES
438
CREATE BLANK TILE
1078
READ UNCOMP FROM DISK
440
CREATE TILE FROM COMPRESSED VERSION
1074
ERROR ?
NO
YES
1080
UNCOMP VERSION VALID ?
NO
YES
1082
INCREMENT UNCOMP LOCK COUNT
1084
LOCKED FOR WRITING ?
YES
NO
1086
INVALIDATE "BLANK" STATUS FLAG
1088
INVALIDATE DISK-RESIDENT UNCOMP VERSION
1090
INVALIDATE & FREE COMPRESSED VERSION
1092
MOVE TO FRONT OF UNCOMP TILE MOST RECENTLY USED LIST
1068
END

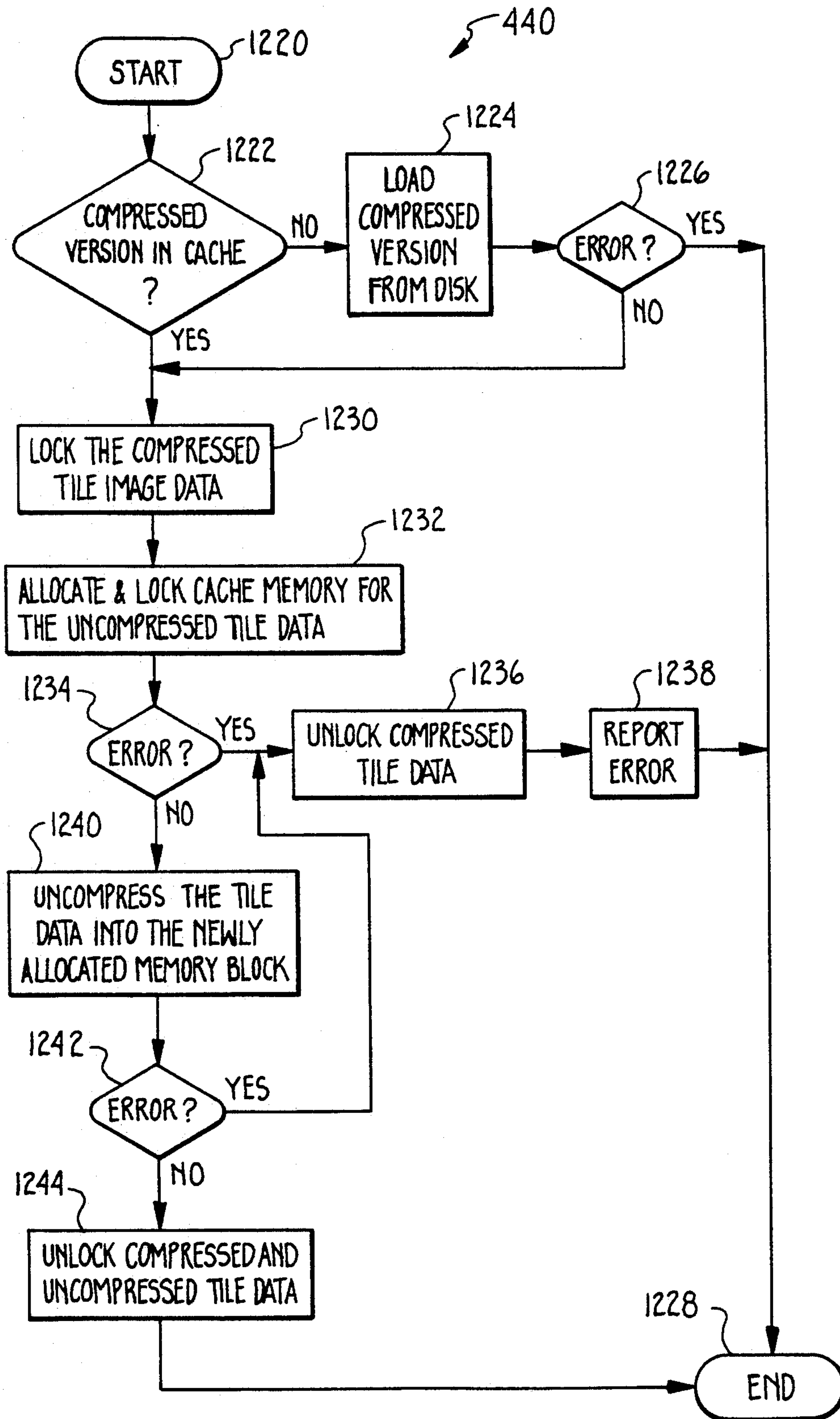

ExpTileUnlock
Fig. 29
430
START
1100
1102
LOOP
1104
LOOP
432
FOR EACH ROW TO UNBLOCK
FOR EACH TILE COLUMN TO UNBLOCK
UNLOCK UNCOMPRESSED VERSION OF TILE
1108
END


Unlock Exp Handle
Fig. 30
432
START
1110
UNCOMP VERSION LOCKED ?
1112
NO
YES
DECREMENT LOCK COUNT
1116
UPDATE OVERVIEWS ?
1118
YES
UPDATE CORRESPONDING LOW-RES TILES
1120
NO
TILE LOCK COUNT=0 ?
1122
YES
CLEAR "CACHE COLLECTION DELAY FLAG"
1124
NO
FREE UNCOMP TILE VERSION
1128
YES
UNCOMP VERSION INVALID ?
1126
NO
END
1114

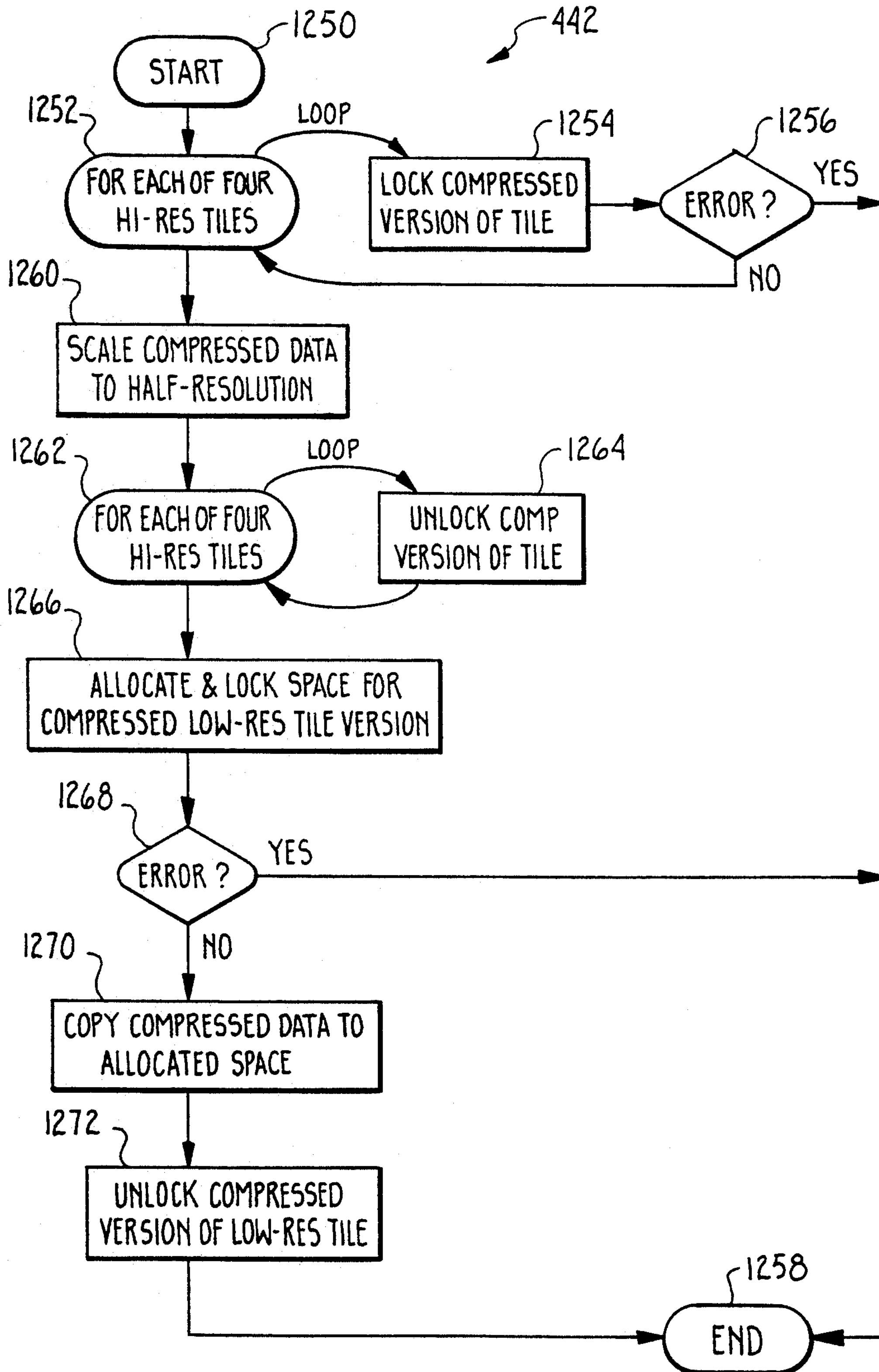

LoadSubImTile
Fig.31
436
START
1140
TILE ALREADY LOADED ?
1142
YES
NO
HIGHER RESOLUTION SUB-IMAGE EXISTS ?
1146
NO
REPORT ERROR
1148
YES
LOCATE THE FOUR HIGHER-RES TILES THAT REDUCE TO THIS TILE
1150
FOR EACH OF FOUR HI-RES TILES
1152
LOOP
HI-RES TILE LOADED ?
1154
NO
CREATE HI-RES TILE FROM HIGHER-RES TILES (RECURSION)
1156
YES
ALL HI-RES TILES BLANK ?
1158
YES
MARK TILE BLANK
1160
NO
FASTER TO SCALE COMPRESSED DATA ?
1162
NO
ALLOCATE MEMORY FOR UNCOMP LOW-RES TILE
1166
YES
CREATE COMPRESSED LOW-RES TILE DIRECTLY FROM COMPRESSED HI-RES TILES
442
FOR EACH OF FOUR HI-RES TILES
1168
LOOP
COPY_HI-RES TILE TO LOW-RES TILE
444
ERROR ?
1170
YES
REPORT ERROR
1172
NO
END
1144



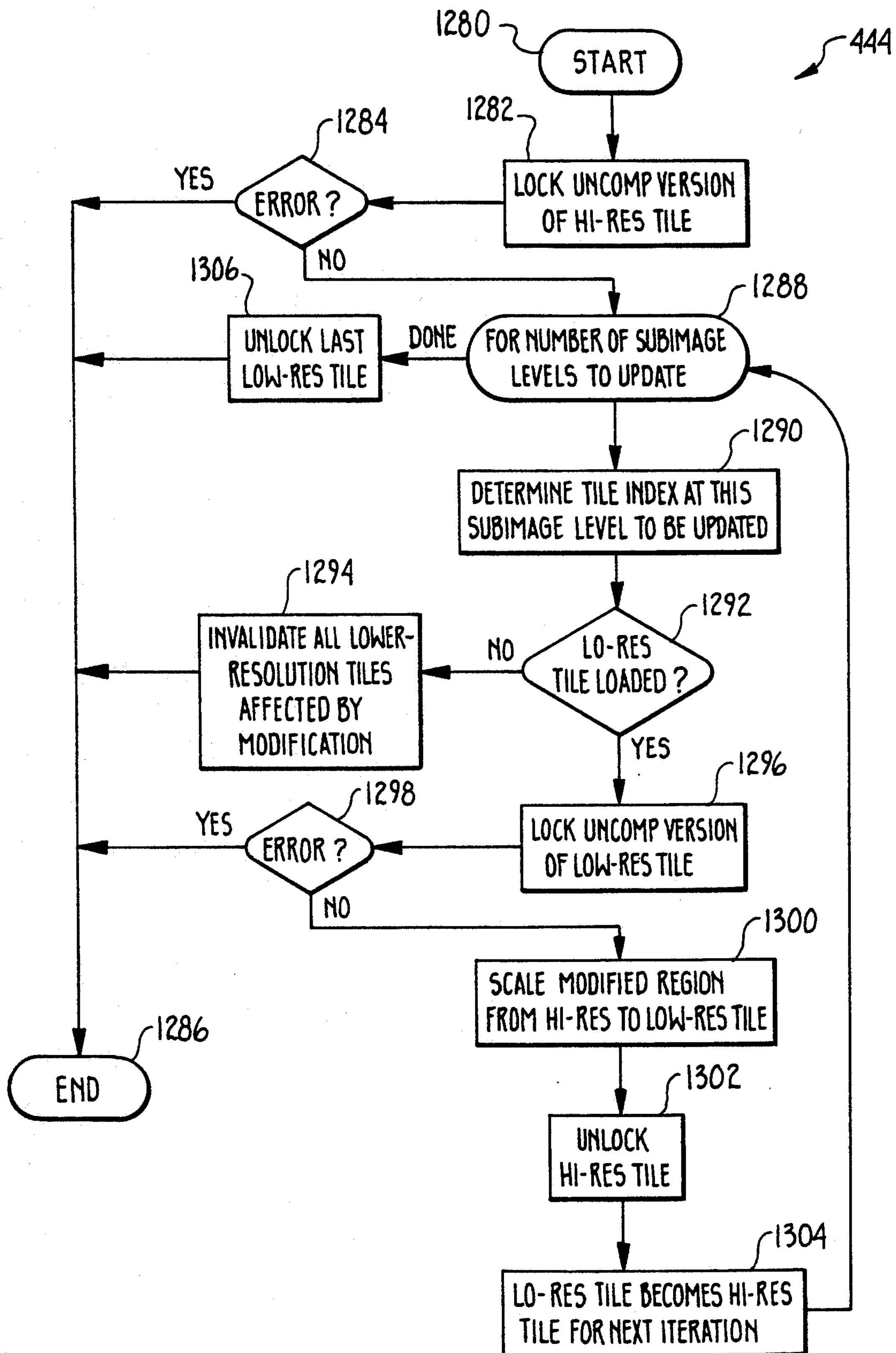
AllocExpHandle
Fig. 32
438
1180
START
1182
SOFT UNCOMPRESSED CACHE USAGE LIMIT EXCEEDED?
YES
1184
"CACHE COLLECTION DELAY FLAG" SET ?
NO
1186
COLLECT FREE CACHE MEMORY BY FREEING UNCOMP TILES
NO
YES
1194
MEMORY AVAILABLE IN UNCOMP TILE FREE LIST ?
YES
NO
1188
SOFT UNCOMPRESSED CACHE USAGE LIMIT STILL EXCEEDED?
NO
YES
1192
SET "CACHE COLLECTION DELAY FLAG"
1190
PRINT WARNING MESSAGE
1196
MEMORY AVAILABLE IN CACHE RESERVE LIST ?
YES
NO
1198
COLLECT FREE CACHE MEMORY BY FREEING COMPRESSED AND UNCOMP TILES
1200
MEMORY AVAILABLE IN CACHE RESERVE LIST ?
NO
YES
1206
FIND FREE BLOCK WITH HIGHEST MEMORY ADDRESS
1208
UNLINK FREE BLOCK FROM LINKED LIST
1210
INITIALIZE FREE BLOCK TO ALL ZEROS
1202
PRINT "CACHE OVERFLOW" ERROR MESSAGE
1216
STORE MEMORY ADDRESS IN TILE HEADER
1214
UPDATE TOTAL UNCOMP CACHE MEMORY USAGE
1212
MOVE TO FRONT OF "MOST RECENTLY USED" BLOCK
1204
END

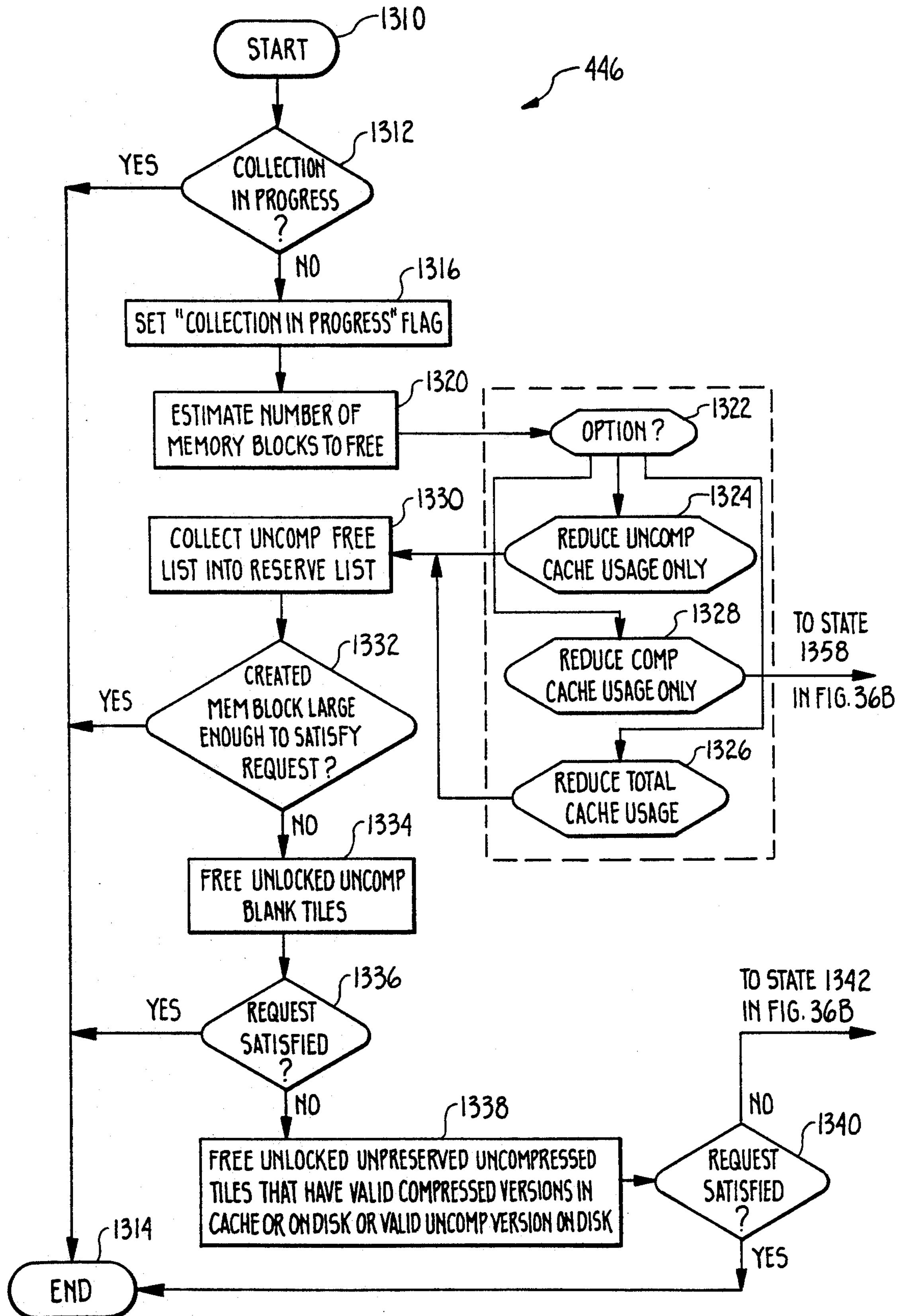


Expand Tile
Fig. 33
440
START
1220
1222
COMPRESSED VERSION IN CACHE ?
NO
YES
1224
LOAD COMPRESSED VERSION FROM DISK
1226
ERROR ?
YES
NO
1230
LOCK THE COMPRESSED TILE IMAGE DATA
1232
ALLOCATE & LOCK CACHE MEMORY FOR THE UNCOMPRESSED TILE DATA
1234
ERROR ?
YES
NO
1236
UNLOCK COMPRESSED TILE DATA
1238
REPORT ERROR
1240
UNCOMPRESS THE TILE DATA INTO THE NEWLY ALLOCATED MEMORY BLOCK
1242
ERROR ?
YES
NO
1244
UNLOCK COMPRESSED AND UNCOMPRESSED TILE DATA
1228
END



 Unified Patents Exhibit 1005 App'x A-N


Copy Tile To Oview
Fig. 35
1280
START
444
1282
LOCK UNCOMP VERSION OF HI-RES TILE
1284
ERROR ?
YES
NO
1306
UNLOCK LAST LOW-RES TILE
DONE
1288
FOR NUMBER OF SUBIMAGE LEVELS TO UPDATE
1290
DETERMINE TILE INDEX AT THIS SUBIMAGE LEVEL TO BE UPDATED
1294
INVALIDATE ALL LOWER-RESOLUTION TILES AFFECTED BY MODIFICATION
NO
1292
LO-RES TILE LOADED ?
YES
1296
LOCK UNCOMP VERSION OF LOW-RES TILE
1298
YES
ERROR ?
NO
1300
SCALE MODIFIED REGION FROM HI-RES TO LOW-RES TILE
1286
END
1302
UNLOCK HI-RES TILE
1304
LO-RES TILE BECOMES HI-RES TILE FOR NEXT ITERATION


Collect Free Cache
Fig. 36A
1310
START
446
1312
COLLECTION IN PROGRESS ?
YES
NO
1316
SET "COLLECTION IN PROGRESS" FLAG
1320
ESTIMATE NUMBER OF MEMORY BLOCKS TO FREE
OPTION ?
1322
1324
REDUCE UNCOMP CACHE USAGE ONLY
1330
COLLECT UNCOMP FREE LIST INTO RESERVE LIST
1328
REDUCE COMP CACHE USAGE ONLY
TO STATE 1358 IN FIG. 36B
1332
CREATED MEM BLOCK LARGE ENOUGH TO SATISFY REQUEST ?
YES
1326
REDUCE TOTAL CACHE USAGE
NO
1334
FREE UNLOCKED UNCOMP BLANK TILES
1336
REQUEST SATISFIED ?
YES
NO
TO STATE 1342 IN FIG. 36B
NO
1338
FREE UNLOCKED UNPRESERVED UNCOMPRESSED TILES THAT HAVE VALID COMPRESSED VERSIONS IN CACHE OR ON DISK OR VALID UNCOMP VERSION ON DISK
1340
REQUEST SATISFIED ?
YES
1314
END

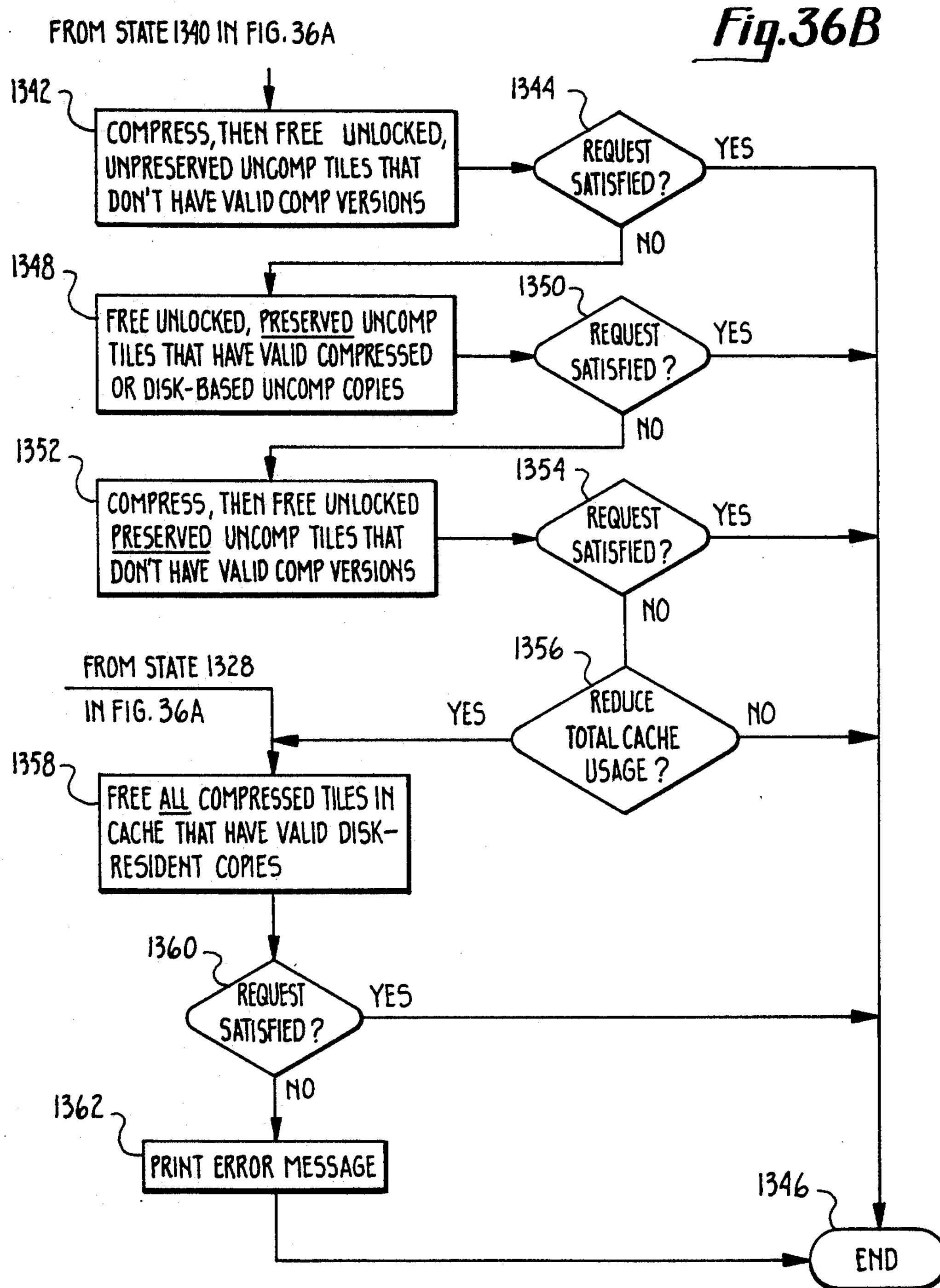
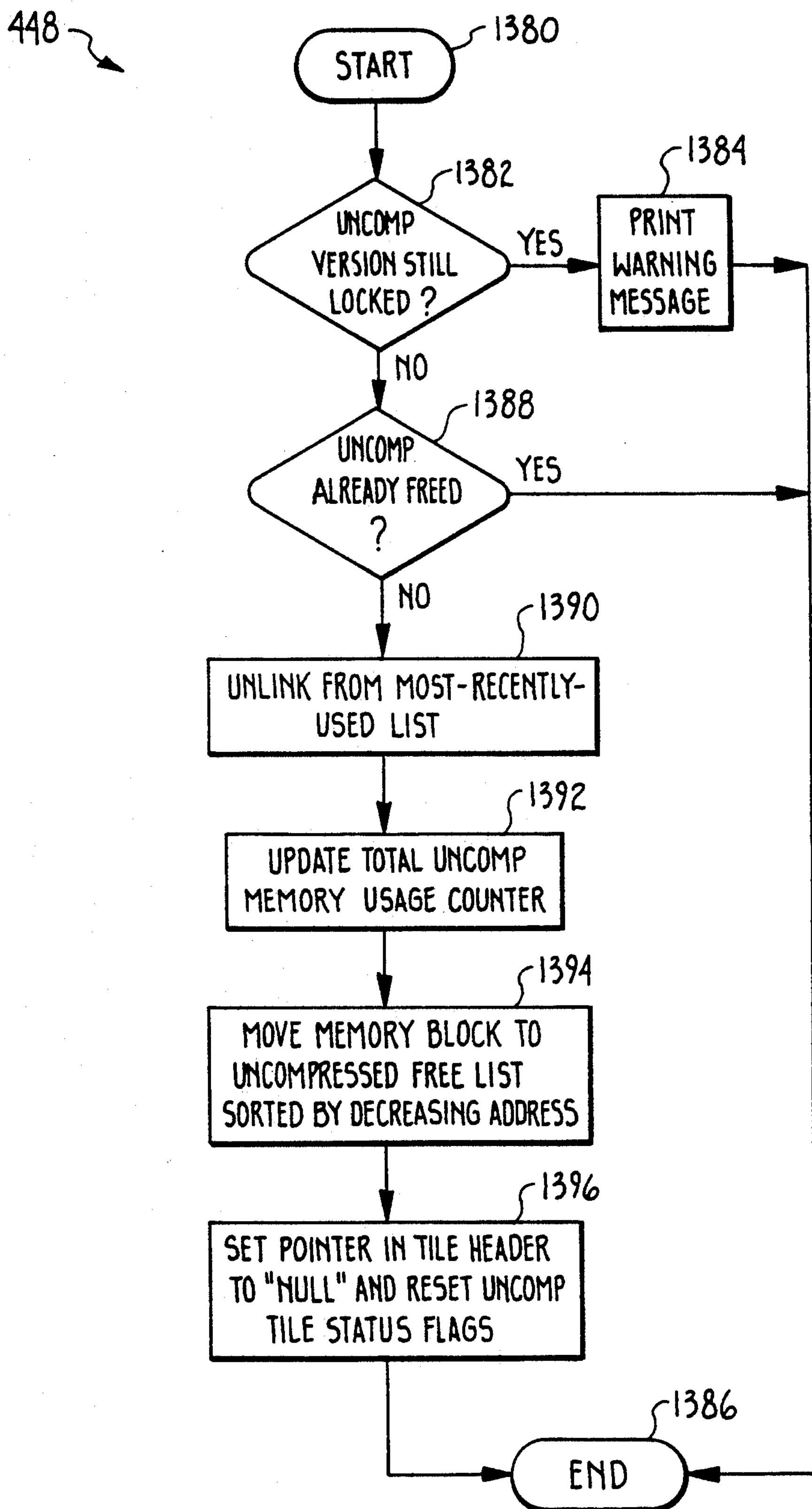
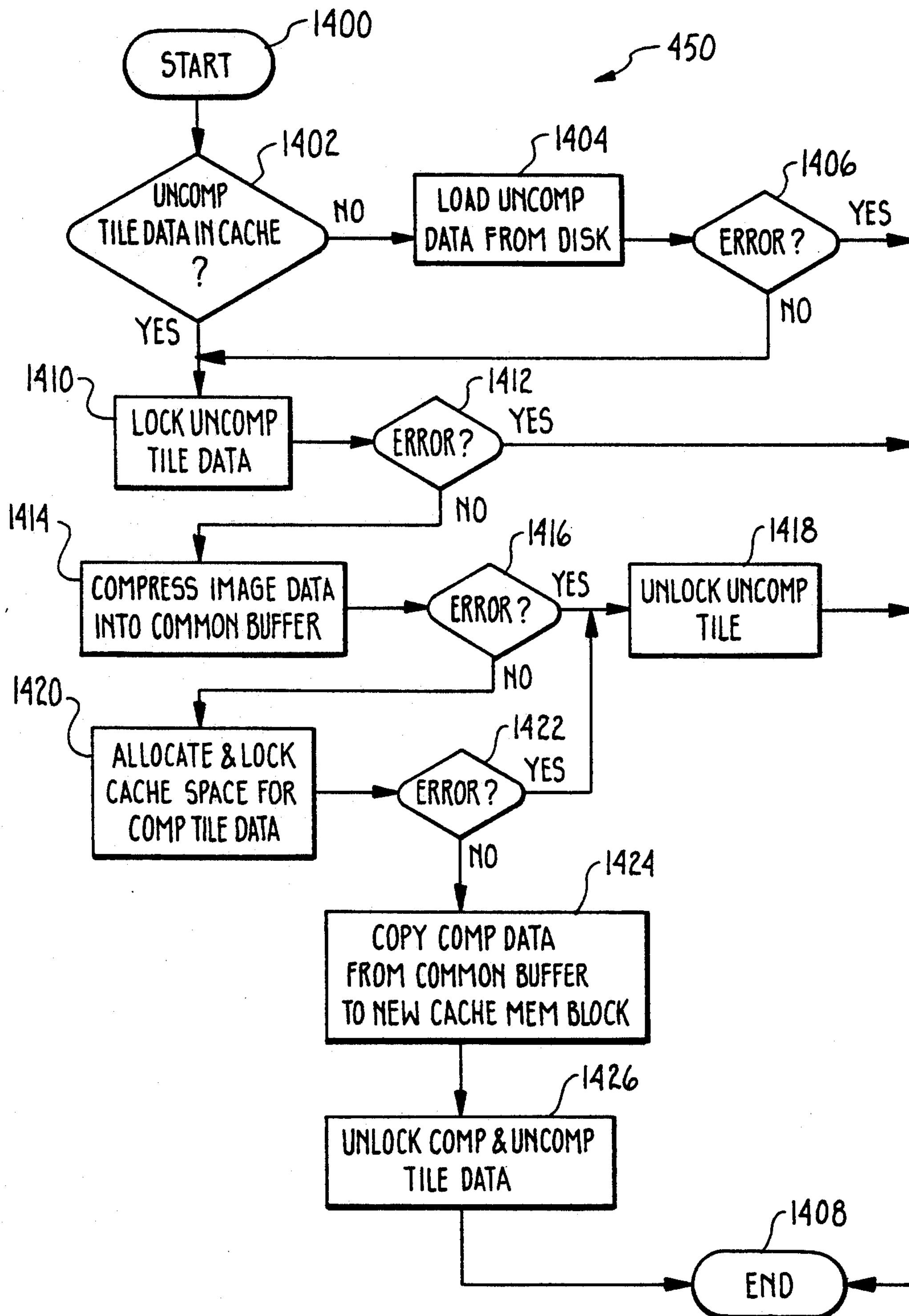


Fig.36B
FROM STATE 1340 IN FIG. 36A
1342
COMPRESS, THEN FREE UNLOCKED, UNPRESERVED UNCOMP TILES THAT DON'T HAVE VALID COMP VERSIONS
1344
REQUEST SATISFIED?
YES
NO
1348
FREE UNLOCKED, PRESERVED UNCOMP TILES THAT HAVE VALID COMPRESSED OR DISK-BASED UNCOMP COPIES
1350
REQUEST SATISFIED?
YES
NO
1352
COMPRESS, THEN FREE UNLOCKED PRESERVED UNCOMP TILES THAT DON'T HAVE VALID COMP VERSIONS
1354
REQUEST SATISFIED?
YES
NO
1356
FROM STATE 1328 IN FIG. 36A
REDUCE TOTAL CACHE USAGE?
YES
NO
1358
FREE ALL COMPRESSED TILES IN CACHE THAT HAVE VALID DISK-RESIDENT COPIES
1360
REQUEST SATISFIED?
YES
NO
1362
PRINT ERROR MESSAGE
1346
END


FreeExpHandle
Fig. 37
448
START
1380
UNCOMP VERSION STILL LOCKED ?
1382
YES
PRINT WARNING MESSAGE
1384
NO
UNCOMP ALREADY FREED ?
1388
YES
NO
UNLINK FROM MOST-RECENTLY-USED LIST
1390
UPDATE TOTAL UNCOMP MEMORY USAGE COUNTER
1392
MOVE MEMORY BLOCK TO UNCOMPRESSED FREE LIST SORTED BY DECREASING ADDRESS
1394
SET POINTER IN TILE HEADER TO "NULL" AND RESET UNCOMP TILE STATUS FLAGS
1396
END
1386


CompressTile
Fig.38
450
START
1400
UNCOMP TILE DATA IN CACHE ?
1402
NO
YES
LOAD UNCOMP DATA FROM DISK
1404
ERROR ?
1406
YES
NO
LOCK UNCOMP TILE DATA
1410
ERROR ?
1412
YES
NO
COMPRESS IMAGE DATA INTO COMMON BUFFER
1414
ERROR ?
1416
YES
NO
UNLOCK UNCOMP TILE
1418
ALLOCATE & LOCK CACHE SPACE FOR COMP TILE DATA
1420
ERROR ?
1422
YES
NO
COPY COMP DATA FROM COMMON BUFFER TO NEW CACHE MEM BLOCK
1424
UNLOCK COMP & UNCOMP TILE DATA
1426
END
1408

SYSTEM FOR MANAGING TILED IMAGES USING MULTIPLE RESOLUTIONS

MICROFICHE APPENDIX

A microfiche appendix containing computer source code is attached. The microfiche appendix comprises one (1) sheet of microfiche having 74 frames.

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates to memory management systems and, more particularly, to the memory management of large digital images.

2. Description of the Prior Art

The present invention comprises a memory management system for large digital images. These digital, or raster, images are made up of a matrix of individually addressable pixels, which are ultimately represented inside of a computer as bit-maps. Large digital images, such as those associated with engineering drawings, topographic maps, satellite images, and the like, are often manipulated by a computer for the purpose of viewing or editing by a user. The size of, such images are often on the order of tens and even hundreds of Megabytes. Given the current cost of semiconductor memory it is economically impracticable to dedicate a random access memory (RAM) to storing even a single large digital image (hereinafter just referred to as a "digital image"). Thus, the image is usually stored on a slower, secondary storage medium such as a magnetic disk, and only the sections being used are copied into main memory (also called RAM memory).

However, as is well known by users of computer aided design ("CAD") systems, a simplistic memory transfer scheme will cause degraded performance during many typical operations, including zooming or panning. Essentially, during such operations, the computer cannot transfer data between disk and main memory fast enough so that the user must wait for a video display to be refreshed. Clearly, these periods of waiting on memory transfers are wasteful of engineering time.

Presently, to enhance main memory storage of only relevant sections of a digital image, the image is logically segmented into rectangular regions called "tiles". Two currently preferred standards for segmenting an image into tiles are promulgated by the Computer Aided Logistics Support (CALS) organization of the United States government (termed the "CALS standard" herein) and by Aldus Corporation of Seattle, Washington, as defined in the Tagged Image Format File (TIFF) definition (e.g., "TIFF Specification, Revision 5.0, Appendix L). Among other tile sizes, both standards define a square tile having dimensions of 512×512 pixels. Thus, if each pixel requires one byte of storage, the storage of one such tile would require a minimum of 256 kilobytes of memory.

Others, such as Thayer, et al. (U.S. Pat. No. 4,965,751) and Sawada, et al. (U.S. Pat. No. 4,920,504) have discussed tiling or blocking a memory. However, such computer hardware is generally associated with a graphics board for improving the speed of pixel transfers between a frame buffer and a video display by addressing a group of pixels simultaneously. These systems have no relationship to tiling of the image itself and thus do not require knowledge of image size. Tiling has also been used to refer to polygon filling as in Dalrymple, et al. (U.S. Pat. No. 4,951,230), which is unrelated to the notion of tiling discussed herein.

The patent to Ewart (U.S. Pat. No. 4,878,183) discusses interlaced cells, each cell containing one or more pixels, for storing continuous tone images such as photographs. The variable size cells are used to vary the resolution of an image according to a distance which is to be perceived by a user. However, the Ewart disclosure does not discuss rasterized binary images containing line drawings, nor does Ewart discuss virtual memory management for modifying or editing images, as will be more fully discussed below.

Even when stored in a mass storage system, an image library, containing a number of digital images, will consume disk space very quickly. Furthermore, "raw" digital images are generally too large to transfer from mass storage to portable floppy disks, or between computer systems (by telephone, for example), in a timely and inexpensive manner unless some means is used to reduce the size of the image. Hence, users of binary images employ image compression techniques to improve storage and transfer efficiencies. One existing compression standard applicable to facsimile transmission, CCITT Group IV, or T.6 compression, is now being used for digital images. Like many other compression techniques, however, the CCITT standard uses statistical techniques to compress data and, hence, it does not always produce a compressed image that is smaller than the original, uncompressed image. That means that image libraries will often contain a mix of compressed and uncompressed binary images. Similar compression standards exist for color and gray-scale images such as those promulgated by the JPEG (Joint Photog. Exp. Group) Standards Committee of the CCITT as SGV III Draft Standard.

At the present time, digital images are typically viewed and modified with an image editor using an off-the-shelf computer workstation. These workstations usually come with a sophisticated operating system, such as UNIX, that employs a virtual memory to effectively manage memory accesses in secondary and main memories. In an operating system having virtual memory, the data that represents the executable instructions for a program or the variables used by that program do not need to reside entirely in main memory. Instead, the operating system brings portions of the program into main memory only as needed. (The data that is not stored in main memory being stored on magnetic disk or other like nonvolatile memory.) The address space that is available to any one application program is generally managed in blocks of convenient sizes called "pages" or "segments".

In general, a virtual memory system allows application programs to be written and executed without concern for the management of virtual memory carried out by the operating system. Thus, independence of the size of main memory is achieved by creating a "virtual" address space for the program. The operating system translates virtual addresses into physical addresses (in a main or cache memory) with the aid of an "address translation table". This table contains one entry per virtual memory segment of status information. For instance, segment status will commonly include information about whether a segment is currently in main memory, when a segment was last used, a disk address at which the disk copy of the segment resides, and a RAM address at which the segment resides (only valid if the segment is currently loaded in main memory).

When the program attempts to access data in a segment that is not currently resident in main memory, the operating system reads the segment from disk into main memory. The operating system may need to discard another segment to make room for the new one (by overwriting the area of main memory occupied by the old segment), so some method of determining which segment to discard is required. Usually the method is to discard the least recently used segment. If the discarded segment was modified then it must be written back to disk. The operating system completes the "swap" operation by updating the address translation table entries of the new and discarded segments.

In summary, the conventional memory management schemes consider data to be in one of two states: resident or not resident in main memory. Which segments are stored in main memory at any given time is generally determined only by past usage, with no way of predicting future memory demands. For instance, just because a segment is the least recently used does not mean that it will not be used at the very next memory access.

However, the management of virtual memory for images departs significantly from conventional virtual memory schemes because images and computer programs are accessed in very different ways. Computer programs tend to access one small neighborhood of virtual address, and then jump to some distant, essentially random, location. However, during normal image processing operations an image is accessed in one of a finite set of predictable patterns. It is not surprising then that conventional memory management systems can significantly degrade performance when used in image processing applications by applying inappropriate memory management rules. Rules which should be abided by a memory management system for large digital images are the following:

1. Image memory must be managed as rectangular image regions (called "tiles"), not as linear memory address ranges.

2. An image tile can exist in five forms: uncompressed memory-resident, compressed memory-resident, uncompressed disk-resident, compressed disk-resident and "can be derived from other available image tiles", in contrast to the two basic forms of memory-resident and disk-resident available in conventional virtual memory schemes.

3. The image region that will be affected by a particular image processing operation is known before the operation begins, and that information can be conveyed to the memory manager.

4. An image memory manager must be tunable to different system capabilities and image types. For example, many computers can decompress a tile of binary data much faster that they can retrieve the uncompressed version of the same tile from disk. On the other hand, some images cannot be compressed at all.

5. An image memory management system should support the capability to "undo" editing operations which is built into the memory manager for optimal performance and ease of use. Thus, the memory manager could easily save copies of the compressed tiles in the affected region, and quickly restore the image to the original state by simply modifying the tile directory entries to point to the old version.

Reader, et al., ("Address Generation and Memory Management for Memory Centered Image Processing Systems", SPIE, Vol. 757, Methods for Handling and Processing Imagery, 1987) discuss a primitive memory management system for images. However, in that system, image tiles are only stored in memory and not on disk. Furthermore, in the Reader, et al., system, there is no capability to handle images in compressed form, nor is there any discussion of "undoing" editing operations.

Consequently, a need exists for an image memory management system that provides: linkages with a raster image editor which includes modify and undo operations, true virtual memory for large images specifying locations on disk and in memory, simultaneous handling of compressed and uncompressed images, and a method for rapidly constructing reduced resolution views of the image for display. The latter need is particularly important when viewing a large image reduced to fit on a video display.

SUMMARY OF THE INVENTION

The above-mentioned needs are satisfied by the present invention which includes a memory management system for tiled images. The memory management system includes a tile manager for maintaining a virtual memory comprising a main memory and a secondary memory such as a disk. The tiled images may include tiles in compressed or uncompressed form.

The tile manager selects the form of image tile that most appropriately matches a request. Each tile of an image may exist in one or more of five different forms, or states, as follows: uncompressed and resident in the image data cache, compressed and resident in the image data cache, uncompressed and resident on disk, compressed and resident on disk and not loaded but re-creatable using data from higher-resolution image tiles.

An image stack having successively lower-resolution subimages is constructed from a full resolution source image. The lower-resolution images in the image stack may be used to enhance such standard image accesses as zooming and panning where high speed image reduction is advantageous.

The image memory management system provides linkages with image processing applications that facilitate image modifications. The tile manager need only store compressed tiles that relate to so-called undoable operations.

These and other objects and features of the present invention will become more fully apparent from the following description and appended claims taken in conjunction with the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a perspective view of an image stack comprising full, half, quarter and eighth resolution tiled images;

FIG. 2 is a full resolution image of a mechanical part;

FIG. 3 is a half resolution image of the mechanical part shown in FIG. 2;

FIG. 4 is a quarter resolution image of the mechanical part shown in FIG. 2;

FIG. 5 is an eighth resolution image of the mechanical part shown in FIG. 2;

FIG. 6 is a block diagram showing one preferred embodiment of a computer system that includes the present invention;

FIG. 7 is a memory map showing the general arrangement of cache memory according to the present invention;

FIG. 8 is a state diagram defining the flow of tile data between different storage states according to the present invention;

FIGS. 9A and B are a diagram of one preferred data structure defining document information according to the present invention;

FIG. 10 is a diagram of one preferred data structure defining a tile header for maintaining the status of compressed or uncompressed tiles;

FIG. 11 is a diagram of a partial calling hierarchy for the various functions of the presently preferred embodiment of the tile manager of the present invention;

FIG. 12 is a flow diagram of one preferred embodiment of the tile manager;

FIG. 13 is a flow diagram defining the "initialize cache manager" function referred to in the flow diagram of FIG. 12;

FIG. 14 is a state diagram of the locking and unlocking of a memory, state, according to the present invention;

FIGS. 15A, 15B, and 15C are a flow diagram defining the "create image access context" function referred to in FIG. 12;

FIG. 16 is a diagram, of a data structure defining the access context referred to in FIGS. 15A,B;

FIGS. 17A and 17B are a flow diagram defining the "save region for undo" function referred to in FIG. 15B;

FIG. 18 is a flow diagram defining the "load tiled raster image" function referred to in FIG. 12;

FIG. 19 is a flow diagram defining the "load TIFF subimage tile information into tile headers" function referred to in FIG. 18;

FIG. 20 is a flow diagram defining a "store tile info in tile headers" function referred to in FIG. 12;

FIG. 21 is a flow diagram defining the "begin undoable raster operation" function referred to in FIG. 12;

FIGS. 22A and 22B are a flow diagram defining the "read rows from region" function referred to in FIG. 12;

FIGS. 23A and 23B are a flow diagram defining the "write rows to region" function referred to in FIG. 12;

FIG. 24 is a flow diagram defining the "close image access context" function referred to in FIG. 12;

FIGS. 25A and 25B are a flow diagram defining the "undo previous raster operations" function referred to in FIG. 12;

FIG. 26 is a flow diagram defining the "quit cache manager" function referred to in FIG. 12;

FIG. 27 is a flow diagram defining the "lock expanded image tile group" function referred to in FIG. 22A;

FIG. 28 is a flow diagram defining the "lock expanded tile" function referred to in FIG. 27;

FIG. 29 is a flow diagram defining the "unlock expanded image tile group" function referred to in FIG. 27;

FIG. 30 is a flow diagram defining the "unlock expanded tile" function referred to in FIG. 29;

FIG. 31 is a flow diagram defining the "create tile from higher-resolution tiles" function referred to in FIG. 28;

FIG. 32 is a flow diagram defining the "allocate space for uncompressed version of tile" function referred to in FIG. 28;

FIG. 33 is a flow diagram defining the "create uncompressed version of tile from compressed version" function referred to in FIG. 28;

FIG. 34 is a flow diagram defining the "create compressed low resolution tile from compressed higher-resolution tiles" function referred to in FIG. 31;

FIG. 35 is a flow diagram defining the "copy uncompressed high resolution tile to uncompressed low resolution tiles" function referred to in FIG. 31;

FIGS. 36A and 36B are a flow diagram defining the "collect freeable cache memory" function referred to in FIG. 32;.

FIG. 37 is a flow diagram defining the "free uncompressed version of tile" function referred to in FIGS. 36A,B; and

FIG. 38 is a flow diagram defining the "create compressed version of tile from uncompressed version" function referred to in FIG. 17B.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

Reference is now made to the drawings wherein like parts are designated with like numerals throughout.

FIG. 1 illustrates an image stack, generally indicated at **100**. The design of the image stack **100** is based on the idea that image memory can be managed as small square regions, called tiles, that are mostly independent of one another. In general, a tile may be either uncompressed (also termed expanded) or compressed. While the basic uncompressed tile size could be a variable, it is presently preferred to be fixed at 32 kilobytes, or 512 pixels by 512 pixels to conform with the Computer Aided Logistics Support (CALS) raster file format standard for binary images. (Note that the present invention allows binary and color images to coexist in a common image memory management system.)

In order to compensate for lower performance expected with a virtual memory management system for images, particularly when reducing large portions (by combining pixels) of the image for display, the present invention automatically maintains a series of reduced resolution copies, called subimages, of the full resolution image. Preferably, the resolution (i.e., pixels per inch) of each subimage is reduced by exactly half relative to the next higher-resolution subimage. Thus, the image stack **100** can be visualizing as an inverted pyramid, wherein the images can be stacked beginning with a full resolution subimage (or image) **102** at the top, followed by a half resolution subimage **104**, then a quarter resolution subimage **106**, and an eighth resolution subimage **108**. (In FIG. **1**, the subimages **102–108** are outlined by bolded lines.)

The subimages **102, 104, 106, 108** are superimposed on a set of tiled subimages **110***a*, **110***b*, **110***c*, **110***d*, respectively, defining sets of tiles. The extent of the image stack **100** ends at the resolution that allows the entire subimage to be stored within a single tile **108** (preferably 512×512 pixels square). Each lower-resolution subimage **104–108** is a faithful representation of the full resolution subimage **102** at all times, with the exception of certain times during operations that modify the appearance of the full resolution subimage **102**.

FIG. 2 illustrates an 8½"×11", A-size mechanical drawing (to scale) as the full resolution subimage **102** showing a mechanical part **120***a*. Of course, other larger drawings such as, for example, D-size and E-size may be used by the present invention. Also, other image processing applications besides mechanical drawings may be used with the present invention including electrical schematics, topographical maps, satellite images, hea-

ting/ventilating/air conditioning (HVAC) drawings, and the like.

FIG. 3 illustrates the corresponding half resolution subimage 104 showing the half resolution part 120*b*. FIG. 4 illustrates the corresponding quarter resolution subimage 106 showing the quarter resolution part 120*c*. Lastly, FIG. 5 illustrates an eighth resolution subimage 108 showing the eighth resolution part 120*d*. In the preferred embodiment, reduced resolution subimages can be used any time that a reduction factor of 2:1 or higher would be used to scale a region of interest in the full resolution subimage 102 for display, plotting or copying.

The subimages 102–108 can be loaded from a source image file, if they exist, or they can be created on demand by the image memory management system of the present invention. The present invention includes editing capabilities that allow a user to trade off between "quick flash" pan/zoom performance and file size as measured by the number of reduced resolution subimages stored with each image. Depending on the application, the user will normally opt to store one or more reduced resolution subimages with each source image file.

The lower-resolution subimages, for example, subimages 104–108, are utilized by the image memory management system to produce the illusion of instant access to any region of the image at any scale factor (not just the scale factor of the overview subimage). Increasing the number of lower-resolution subimages gives a higher quality "first flash" image during panning and zooming and reduces the time to get the final version of the image to the screen.

FIG. 6 illustrates a computer workstation generally indicated at 150 which is representative of the type of computer that is used with the present invention. The workstation 150 comprises a computer 152, a color monitor 154, a mouse 156, a keyboard 158, a floppy disk drive 160, a hard disk drive 162 and an Ethernet communications port 164. The computer 152 includes a motherboard bus 166 and an I/O bus 168. The I/O bus 168, in one preferred embodiment, is an IBM PC/AT ® bus, also known as an Industry Standard Architecture (ISA) bus. The two buses 166, 168 are electrically connected by an I/O bus interface and controller 170.

The I/O bus 168 provides an electromechanical communication path for a number of I/O circuits. For example, a graphics display controller 172 connects the monitor 154 to the I/O bus 168. In the presently preferred embodiment, the monitor 154 is a 19-inch color monitor having a 1,024×768 pixel resolution. A serial communications controller 174 connects the mouse 156 to the I/O bus 168. The mouse 156 is used to "pick" an image entity displayed on the monitor 154.

The I/O bus 168 also supports the hard disk drive 162, and the Ethernet communications port 164. A hard disk controller 176 connects the hard disk drive 162 to the I/O bus 168. The hard disk drive 162, in one possible configuration of the workstation generally indicated at 150, stores 60 megabytes of data. An Ethernet communications controller 178 connects an Ethernet communications port 164 with the I/O bus 168. The Ethernet communications controller 178 supports the industry standard communications protocol TCP/IP which includes FTP and Telnet functions. The Ethernet communications port 164 of the preferred embodiment allows the Workstation 150 to be connected to a network which may include, among other things, a document scanner (not shown) and a print server (not shown).

The motherboard bus 166 also supports certain basic I/O peripherals. For example, the motherboard bus 166 is connected to a keyboard and floppy disk controller 180 which supports the keyboard 158 and the floppy disk drive 160. The floppy disk drive 160, in one present configuration, can access floppy disks which store up to 1.2 megabytes of data.

The fundamental processing components of the computer 152 are a microprocessor 182 such as, for example, an 80386 microprocessor manufactured by Intel, a math coprocessor 184 such as, for example, a 80387 math coprocessor also manufactured by Intel and a main memory generally indicated at 186 comprising, for example, 4 megabytes of random access memory (RAM). The main memory 186 is used to store certain computer software including a Unix compatible operating system 188 such as, for example, SCO Xenix licensed by Santa Cruz Operation of Santa Cruz, California, a subsidiary of Microsoft Corporation, an image processing application 190, a tile manager 192, and an image data cache 194. The image processing application 190 includes editing functions such as zoom and pan.

Another presently preferred computer workstation 150 having somewhat different processing components from those just described is available from Sun Microsystems, Inc. of Mountain View, California, under the tradename "SPARCstation 1". In such an embodiment, the UNIX compatible operating system would be licensed directly from Sun.

Although a representative workstation has been shown and described, one skilled in the applicable technology will understand that many other computer and workstation configurations are available to support the present invention.

FIG. 7 illustrates a representative configuration of the image data cache 194 some time after the tile manager 192 (FIG. 6) begins operation. A set of compressed tiles 222 are kept at the low addresses of the image data cache 194, and a set of uncompressed (or expanded) tiles 224 at the high addresses of the image data cache 194. The terms expanded or uncompressed are used interchangeably. In between the two sets of tiles 222, 224 is a reserved area 226 (free cache memory). As the operation of the tile manager 192 continues, the image data cache 194 becomes more unordered. As the cache requirement for compressed or uncompressed tiles increases, each set of tiles 222, 224 approach the reserve area 226 from each end. In fact, the reserve area 226 can become completely exhausted.

Since the memory management schemes that apply to compressed data allocation are very different from that of uncompressed data, it is desirable to keep the two sets of tiles 222, 224 separate. Compressed tiles are variable sized tiles (blocks of memory) 222*a,b,c,d,e,f* whereas the uncompressed tiles are all fixed sized tiles 224*a,b,c,d* and therefore the locations of the fixed sized tiles 224 are interchangeable. Linked lists of allocated memory are kept sorted according to size and address for compressed tiles. The number of linked lists is a variable number but presently there are about 64 different size categories for compressed tiles and only one size category for uncompressed tiles (for binary images).

To use the image data cache 194, the memory management functions begin by determining how much fast memory (RAM) and slow memory (disk or host memory) is available for image memory uses. When an image

is loaded, the system allocates memory for image information and related tile directory structures. Cache management parameters are modified as necessary to balance the requirements for expanded tile and compressed tile cache memory. The expanded tile cache memory pool and the compressed tile cache memory pool allow tiles from different images to intermingle. Expanded and compressed tiles are kept in separate areas as much as possible so that memory allocation can be optimized for each of two different situations (i.e., fixed allocation block size versus variable size). However, the storage ranges of compressed and expanded tiles are allowed to mingle so as to maximize the flexibility of the cache usage.

FIG. 8 is a state diagram illustrating the flow of image data or tiles between different storage states 250. A tile can contain data in one or more of five states or forms as illustrated by ovals in FIG. 8. The possible forms are: uncompressed and resident in cache memory (state 252); compressed and resident in cache memory (state 256); uncompressed and resident on disk (state 268); compressed and resident on disk (state 262); "not loaded" but re-creatable using information from higher-resolution image tiles (state 272).

For most image access operations, the image data must be uncompressed and resident in cache memory 252. However, that form consumes the most cache memory of any of the five forms. Therefore, a primary function of the tile manager 192 is to transform image tile data between state 252 and the other states which consume less (in the case of state 256) or no cache memory whatsoever (in the cases of states 268, 262 and 272).

The eight transformation operations, shown in square boxes in FIG. 8, constitute the main computational operations associated with managing image memory. The operation "load compressed tile image data from disk into cache memory" 264 is typically the first operation performed on a tile because most pre-scanned images are stored in compressed form in disk files. (A discussion of this "virtual loading" is provided hereinbelow.) The load operation 264 is performed by the Load CompFromDisk function which simply copies data from the disk into cache memory. The disk location and number of bytes to read is stored in the tile header fields 368 and 376 shown in FIG. 10.

The function LoadCompFromDisk is normally used by the function LockCompHandle when the tile manager 192 needs to access the compressed form of data associated with a tile. LockCompHandle is analogous to LockExpHandle, described in FIG. 28. The LockCompHandle function is also included in source code form in the Microfiche Appendix, in the file tilealloc.c.

Compressed data in cache 256 can be written back to the disk by the operation 260. This is the reverse of the LoadCompFromDisk function. The present embodiment is capable of writing to disk in a wide variety of file formats. One skilled in the art can easily create a function to perform this task.

Compressed data in cache can be uncompressed (also termed "expanded") into another region of cache memory by the expand operation 258. The expand operation 258 is controlled by the "Expand Tile" function 440 which is described with respect to FIG. 33. The method of image compression varies according to image type (e.g. binary, 8-bit color, 24-bit color). Commonly used compression techniques include CCITT T.6 for binary images and CCITT SGVIII (draft standard) for color and gray-scale images. The ExpandTile function 440 selects the appropriate compression algorithm by referring to field 306 of the Document Information Structure shown in FIG. 9.

Uncompressed data in cache 252 can be compressed and written to a separate region of cache memory by the compress operation 254. The compress operation 254 is controlled by the CompressTile function 450 described with respect to FIG. 38. Like ExpandTile, the CompressTile function 450 uses an image compression algorithm appropriate to the image type.

Uncompressed data on disk 268 can also be read directly into cache memory by the load operation 270. The load operation 270 is performed by the LoadExpFromDisk function, which appears in source code form in the Microfiche Appendix, in file diskcach.c. The LoadExpFromDisk function is analogous to LoadCompFromDisk. The LoadExpFromDisk function refers to the fields 362 and 374 of the tile header 350 shown in FIG. 10, for the location and number of bytes of the expanded file data on the disk.

Uncompressed data in cache 252 can be written back to the disk by the save to disk operation 266. This operation is analogous to the save to disk operation 260 which operates on compressed data. The present embodiment can write compressed or uncompressed tile data to disk in a variety of formats. One skilled in the art can easily implement an equivalent function.

Image data for tiles in the "not loaded" state 272 must be constructed by resampling higher-resolution tiles. (During normal operation, only lower-resolution tiles can exist in this state—the full resolution subimage tiles are always "loaded".) The present embodiment provides two operations from the "not loaded" state 272 to the "loaded" state 252, 256. Uncompressed higher-resolution tile data is resampled to create uncompressed data in cache 252 by the resample operation 274. Similarly, in the resample operation 276, compressed data in cache 256 can be created from compressed higher-resolution tile data.

In both resampling operations, extensive advantage is taken of the fact that the resolutions of adjacent subimages in the subimage stack are related by a power of 2. This greatly simplifies and speeds the resampling operation. Basic resampling techniques are well-known (See, for example, A. Rosenfeld and A. C. Kab, *Digital Picture Processing,* Academic Press, 1976). The resampling operation 274 and 276 are controlled by the function LoadSubImTile 436 described with respect to FIG. 31.

In summary, FIG. 8 shows that a great part of the tile manager's utility derives from its ability to coordinate a variety of forms of image data in the course of complex image processing operations.

Generally, the way data starts out on the disk 162 is by loading a tiled image file into an application 190 via the tile manager 192. An image file, like a Tagged Image File Format (TIFF) or CALS tiled image file, for example, can be loaded instantaneously, in a virtual sense. In the tiled formats, there are tiled image data that is stored in the image file and at the beginning of the file there is a directory with entries that locate the tiles (for example, the disk file version of tile 0 in subimage 0, (0,0), is located at one address in the file and the disk file version of tile 1, subimage 0 (0,1) is located at another address in the file). When an image file is loaded, the tile manager 192 gets the tile offsets and stores them in the tile directory and does nothing else. Hence, the image file is basically loaded without copying any data from the disk 162 into the image data cache

194, and a directory is created that maps the tiles in the virtual image memory space onto the disk 162.

FIG. 9A illustrates a document information structure 300. Each image, or document, in the system is associated with (and described by) a document information structure (called "docinfo", defined in FIG. 9). The docinfo structure contains information about the image as a whole, such as color and pixel organization, etc. It also contains a list of subimages contained in the image. Each subimage entry in the docinfo structure contains information about that subimage, such as width and height, etc. The intention is to make this data visible only to cache management functions and low-level access functions. The overall docinfo data structure 300 contains the following information:

- 302 Self-reference to document handle. Handle value assigned to this document by the host procedure which created the document. This value is unique over the entire system.
- 304 "Overviews Invalid" flag. This flag is true if the document is in the middle of a write operation.
- 306 Cache image compression algorithm. Compression algorithm used by the memory manager for this image.
- 308 Image color type. How the image is displayed.
- 310 Bits per image pixel. Number of bits per image pixel.
- 312 Tile size information. Size of expanded tile in pixels. The tiles are assumed to be square.
- 314 Number of subimages in doc. Number of subimages maintained in this document. The minimum value is one (the full resolution subimage).
- 316 Input file info. Input raster file information.
- 318 Output file info. Output raster file information.
- 320 List of subimage headers. Array of pointers to subimage header structures 321. The first entry in the array is always the full resolution image. Each position thereafter corresponds to a 2× resolution reduction from the previous subimage.

The subimage header structure 321 is illustrated in FIG. 9B. Each subimage has its own entry with each field as follows:

- 312 Pointer to tile headers.
- 314 Pointer to tile directory. Pointer to array of pointers to tile header records. This two-dimensional table provides an easy way to access individual tile headers on a (row,col) basis.
- 326 Subimage width and height. The width (x extent) and height (y extent) of the document measured in pixels.
- 328 Number of tile rows & cols in subimage. Number of tile rows in the image and the number of tile columns (i.e., the number of tiles needed to span the height and width of the image).
- 330 Image stack index of this subimage. This is the position of the subimage in the docinfo structure subimage list. It can also be used to determine the factor by which the subimage resolution is reduced relative to the full resolution subimage.
- 332 Pixel resolution of this subimage. Scan resolution in pixels per millimeter.

FIG. 10 illustrates the tile header 350. The tile manager's analog to the conventional address translation table is the tile directory. The tile directory is a two-dimensional array of entries corresponding to the two-dimensional array of tiles that form the image. Each full and reduced resolution image has its own tile directory. The tile directory record contains a list of pointers to lists of individual tile headers. The list in the tile directory record has one entry for each row of tiles. Each of those entries points to a tile header record list with as many elements as tile columns. Thus, there is one tile directory record per subimage and one tile header record per tile. The tile header record defines the current state of the tile and contains information used by the cache management functions. The tile header contains the following information:

- 352 Pointer to document containing this tile. Pointer to the document to which this tile belongs.
- 354 Index of subimage containing this tile. Index of the subimage (i.e., image stack layer) that contains this tile.
- 356 Row and column indices of tile. Tile row and column position of this tile within the subimage.
- 358 Status information. Defines the current state of the tile. This includes lock counts for expanded and compressed tiles.
- 360 Preserve count. Value greater than zero means the tile is desired for future operation, so the tile should be preserved in cache if possible.
- 362 Location of uncompressed image data in cache memory. Location of uncompressed (expanded) image data for this tile (if it exists). Status flag "ExpCached" will be true to indicate that the data is currently in expanded tile cache memory.
- 364 Location of compressed image data in cache memory. Location of compressed image data for this tile (if it exists). Status flag "CompCached" will be true to indicate that the data is currently in compressed tile cache memory.
- 366 Location of uncompressed image data on disk. Location of uncompressed (expanded) image data for this tile (if it exists). Status flag "ExpOnDisk" will be true to indicate that the data is currently on disk.
- 368 Location of compressed image data on disk. Location of compressed image data for this tile (if it exists). Status flag "CompOnDisk" will be true to indicate that the data is currently on disk.
- 370 Link to next less recently used tile. Pointer to next older (less recently used) tile, not necessarily a tile in this image.
- 372 Link to next more recently used tile. Pointer to next newer (more recently used) tile, not necessarily a tile in this image.
- 374 Number of bytes of expanded data in tile.
- 376 Number of bytes of compressed data in tile.

FIG. 11 illustrates a calling hierarchy 400 for the constituent functions. Further discussions relating to flow diagrams, herein, will include names which correspond to source code modules written in the "C" programming language. The object code is presently generated from the source code using a "C" compiler licensed by Sun Microsystems, Inc. However, one skilled in the technology will recognize that the steps of the accompanying flow diagrams can be implemented by using a number of different compilers and/or programming languages.

The top level in the program hierarchy is Main 402. Main initiates the functions calls to the lower level functions. Main embodies the top level control flow of the present invention.

The first function called by Main is Initialize Cache Manager 404 (InitCacheManager). InitCacheManager allocates the RAM and disk swap space needed for a

particular raster image. It must be called before attempting to load any image tiles into memory.

The next function Main may call is Load Tiled Raster Image **408** (LoadTIFF). LoadTIFF manages the loading of tiled images. This is the process where an existing image file on disk is mapped into memory.

Main will then call the function Begin Undoable Raster Operation **410** (BeginUndoableRasOp). BeginUndoableRasOp marks the beginning of a distinct, "undoable" raster image operation. This function does not save any region of image memory but only creates a new entry on the undo stack. The current version of the tiles in the affected region are saved by InitImageAccess.

The following function called by Main is Create Image Access Context **412** (InitImageAccess). InitImageAccess prepares the tile cache manager for upcoming accesses to a particular region of the specified image. This function creates a data structure called an "access context" (defined in FIG. **16**) that is used by the sequential access functions.

Main optionally calls the function Read Rows From Region **414** (ReadRowToRow) next according to the operation performed by the user. ReadRowToRow causes one input/output buffer row or strip to be read and transformed from tiled image memory as specified in the associated InitImageAccess call and the resulting access context.

The next optional function called by Main is Write Rows To Region **416** (WriteRowToRow), again according to the operation performed by the user. WriteRowToRow causes one input/output buffer row or strip to be transformed and written to tiled image memory as specified in the associated InitImageAccess call and the resulting access context.

It should be understood that other access functions, such as random pixel accesses, may optionally be called by Main.

Main then calls the function Close Image Access Context **418** (EndImageAccess). EndImageAccess terminates and discards an image access context. The memory allocated for the access context structure is freed. The tile manager is informed that the specified region of image memory is no longer needed by this operator.

The next function, Undo Previous Raster Operations **420** (UndoPreviousRasOp), is optionally called by Main. UndoPreviousRasOp restores the specified region to its original state using information from the undo stack.

The last function Main calls is Quit Cache Manager **422** (EndCacheManager). EndCacheManager frees the RAM and disk swap space. This function basically reverses what InitCacheManager does.

The second level of functions on the calling hierarchy **400** is shown starting with Load TIFF Subimage Tile Information into Tile Headers **424** (LoadTiffTilesStd) which is called by function LoadTIFF **408**. LoadTiffTilesStd manages the loading of TIFF images with strip structure.

The LoadTiffTilesStd function **424** calls a function Store Tile Information in Tile Headers **425** (LoadSubImDiskCache). LoadSubImDiskCache loads the tile directory of the specified subimage with information about the location, size and format of individual image tiles contained in a disk-resident tiled image file. It is the low-level interface for the "indirect file load" capability. The tile headers are assumed to be completely zeroed when this function is called.

The InitImageAccess function **412** calls a function Save Region For Undo **426** (SaveRegionForUndo). SaveRegionForUndo saves the specified region on the undo stack. It is called from within InitImageAccess if the SaveForUndo flag is true. It can also be used for low level operations that do not go through InitImageAccess. SaveRegionForUndo can then be called multiple times for different documents and different regions within a document so that arbitrarily complex editing operations can be easily undone.

The ReadRowToRow function **414** calls a function Lock Expanded Image Tile Group **428** (ExpTileLock). ExpTileLock "locks" memory handles referring to expanded image tiles. (The notion of locking and unlocking memory blocks is further discussed below with reference to FIG. **14**.) It also updates the associated tile header structure as appropriate for the operating system.

The ReadRowToRow function **414** also calls a function Unlock Expanded Image Tile Group **430** (ExpTileUnlock). ExpTileUnlock unlocks memory handles referring to expanded image tiles. It also updates the associated tile header structure as appropriate for the operating system.

The function ExpTileUnlock **430** calls a function Unlock Expanded Tile **432** (UnlockExpHandle). UnlockExpHandle unlocks an individual expanded tile handle. The lock count is decremented as appropriate. The tile is not actually swapped out of cache at this point but it becomes a candidate for swapping.

The function ExpTileLock **428** calls a function Lock Expanded Tile **434** (LockExpHandle). LockExpHandle locks an individual expanded tile handle. The lock count is incremented and the status flags are set as appropriate.

The LockExpHandle function calls a function Create Tile From Higher-Resolution Tiles **436** (LoadSubImTile). LoadSubImTile creates a valid expanded version of the specified tile by scaling down from the next higher-resolution subimage. This function is called recursively as necessary to get to a higher-resolution subimage where there is valid data. (Note: the tiles in the full-resolution subimage are always valid and loaded although not necessarily present in the cache memory.)

The function LockExpHandle **434** next calls a function Allocate Space for Uncompressed Version of Tile **438** (AllocExpHandle). AllocExpHandle allocates space in cache memory for a single expanded tile.

The function LockExpHandle **434** also calls a function Create Uncompressed Version of Tile From Compressed Version **440** (ExpandTile). ExpandTile uses a tile that exists in compressed form but not expanded form, allocates space for an expanded tile and decompresses the image data into that space.

The function LoadSubImTile **436** calls a function Create Compressed Lower-Resolution Tile From Compressed Higher-Resolution Tiles **442** (CompCopyToOview). CompCopyToOview creates a valid compressed version of the specified tile by scaling down from compressed or expanded version of the given higher-resolution subimage tiles. The function LoadSubImTile **436** also calls a function Copy Uncompressed High-Resolution Tiles to Uncompressed Low-Resolution Tile **444** (CopyTileToOview). CopyTileToOview updates the region of the next low-

er-resolution overview corresponding to the specified tile.

The Function CompCopyToOview **442** calls a function Collect Freeable Cache Memory **446** (CollectFreeCache). CollectFreeCache collects freed memory states or enlarges the cache file and adds the new memory capacity to the reserve list. This function is called when the cache manager usage exceeds preset limits. Therefore it makes sense to take time to free up as much memory as is convenient at this opportunity.

The function CollectFreeCache calls a function Free Uncompressed Version of Tile **448** (FreeExpHandle). FreeExpHandle frees space used for storage of expanded image tiles.

The function CollectFreeCache **446** also calls a function Create Compressed Version of Tile From Uncompressed Version **450** (CompressTile). CompressTile uses a tile that exists in expanded form but not compressed form, allocates space for a compressed tile and compresses the image data into that space.

FIG. **12** is the top-level control flow for the tile manager **192** (also called "Main"). The tile manager **192** can be executed on a number of operating systems or without an operating system. However, the workstation **150** (FIG. **6**) preferably includes the Unix compatible operating system **188**. Another preferred operating system is Microsoft MS-DOS running with or without Microsoft Windows 3.0.

Moving from a start state **470** to an initialization state **404**, the tile manager **192** performs an initialization of the image data cache **194** to determine the available memory space, or the amount of physical RAM and disk space available for a cache "file". At this point, the cache appears to the tile manager **192** as one contiguous range of physical addresses in memory. If the tile cache has already been initialized, this step is skipped. The possibility of multiple image access contexts (discussed below) allows multiple simultaneous requests.

The tile manager **192** has another parameter which is called the fast memory portion of the image data cache **194**. This parameter is particularly relevant when working on top of another virtual operating system such as Unix. The fast memory limit specifies approximately how much of the image cache file is actually kept in RAM memory at any moment by the native operating system (e.g., Unix). The balance of data (the less recently used portion) is likely to have been swapped out to the disk. The tile manager attempts to limit the amount of cache space used to store expanded tiles to less than the fast memory limit, but the limit can be exceeded if necessary with some degradation in performance. However, the total cache size limit is never exceeded. In operating systems without virtual memory capabilities built in (e.g., MS-DOS), the fast memory limit is the same as the total cache size limit.

Then the tile manager **192** moves to a function **472** wherein the tile manager **192** loads a tiled raster image file. The function **472** (comprising the function **408**, for example) loads any type of image file, and preferably a tiled image, into the memory address space configured by the tile manager **192**. If the image to be modified is already loaded, this step is skipped. Then the tile manager **192** moves to a function **410** where the tile manager **192** marks the beginning of an undoable raster operation if the tile manager **192** is writing to the image. The function **410** is an optional state and it is only used if the user wants to be able to undo the operation that modifies the image.

Any time that a region of the image needs to be accessed (for reading or writing) an image access context is created. This image access context is used to define the region for use by the tile manager. The creation is performed automatically by the file manager without effort by the user. For example, an image access context is created when the user draws a line in a region of the image.

Referring back to FIG. **12**, the tile manager **192** transitions to a function **412** to create the image access context. The image access context contains all of the state information about the access operation. It is possible to have multiple access contexts opened simultaneously with each access having stored state information contained in the access context. Thus, the tile manager **192** is re-entered and re-used by interleaved operations without confusion due to the unique access contexts of each image operation.

The tile manager **192** proceeds to a loop state **474** wherein the tile manager **192** begins a FOR-loop for all of the rows or columns in the region. The FOR-loop is executed multiple times if the operation specified by the user is a row or column strip oriented access. Strips are composed of one or more rows or one or more columns of data. For each of the strips, the tile manager **192** reads or writes the rows or columns of data in the strip in a function **476**. The function **476** actually comprises a set of functions including ReadRowtoRow **414** (FIG. **11**) and WriteRowtoRow **416**.

When the tile manager **192** has processed all the row and columns in the region, the tile manager **192** moves to a function (EndImageAccess) **418** where the tile manager **192** closes the image access context which frees all of the temporary buffers that were allocated for the image access context.

The tile manager **192** transitions to an undo previous raster operation function (UndoPreviousRasOp) **420**. This causes a modified image to revert to its previous state. The image tiles that had been modified are replaced by their original versions. This again is an optional step that the user initiates, if a mistake is made.

If the raster image is required for future operations, the tile manager moves to state **422**. Otherwise, moving to a state **478**, the tile manager **192** unloads the raster image. Unloading the raster image simply frees the memory that had been associated with that particular raster image. This is not a save raster image operation which would be slightly more complicated, but a save operation could be executed here. Of course, the image processing application **190** supports loading and saving raster images.

If more operations will be performed the tile manager moves to state **480**. Otherwise, from state **478**, tile manager **192** moves to a quit cache manager function (EndCacheManager) **422**. Herein, the tile manager **192** frees the image data cache **194** (FIG. **6**). Presumably, all of the images have been unloaded as in the state **478** so that this operation frees the image data cache memory and prepares the system for shut down. Lastly, the tile manager **192** terminates at an end state **480**.

FIG. **13** illustrates the initializing of the cache manager function **404**. The function **404** is entered by the task manager **192** at a state **488**. Then, moving to a state **490**, the task manager **192** initializes the cache usage variables. Of course, in the beginning, all of the cache space is available for use, in what is called the free-memory reserve list. That is, no cache memory is being used for expanded or compressed image data.

At state **492**, the task manager **192** allocates tile cache memory by requesting a portion of the address space from the memory space owned by the operating system. In a virtual memory system such as Unix, the request is handled by memory mapping a large file. The operating system does not allocate any memory, but it reserves an address space. Moving to a state **494**, the task manager **192** allocates a common blank tile. When dealing with binary images, space is reserved for one blank tile, which is kept around at all times for common usage by any number of operations, or access contexts.

At state **496** a compression buffer is allocated to be used as a scratch buffer when compressing data since, in general, the size of the resulting compressed data is unknown before a tile of image data is compressed. Hence, compressed data blocks will be variable sized. The tile manager **192** then exits the InitCacheManager function **404** at an end state **498**.

FIG. **14** illustrates a general memory state diagram with reference to a block of memory being "locked" or "unlocked". In the diagram, ovals are states and rectangular blocks are operations.

The state diagram is entered at a start state **502** by a new memory block. There are three basics states. "FREE" is a state **504** where there is no memory allocated. Actually, a block of memory is considered free if it is in one of the memory free lists, i.e., the "reserve free list", the "compressed free list" or the "expanded tile free list". It should be understood that the free list for the compressed tiles are actually composed of many lists based on the varying sizes of memory blocks.

Within a tile header (FIG. **10**) the tile manager **192** controls a memory handle which is a structure that has a pointer to (or location of) image data in the cache and a lock count (not shown) for both compressed and expanded versions of a tile.

A memory block transitions from the free state to unlocked, but allocated is through a state **506** for allocating the memory handle, which moves the block out of the free list and into use by a tile. As opposed to free, unlocked means that the memory block contains valid data and that it is associated with a tile but not currently being accessed. That is, the block is not being read or written at the time.

Now, the tile is unlocked at a state **508** but it contains valid data. Therefore, the next step is to lock the block, or lock the memory handle at a state **512** and then it becomes a locked memory state at a state **514**. That means it contains valid data and it is currently in use. The block can be locked more than once, each time just incrementing the lock count.

The lock count may be incremented multiple times, for example, when two access contexts (operations) are accessing the same region of memory. Hence, both contexts lock the block of memory or tile by incrementing the lock count. When the first access context is done it decrements the lock count. But the tile manager **192** knows that that tile is still in use by an access because the locked count is still non-zero.

The inverse operation is to unlock the handle at a state **516** and as long as the lock count is not decremented to zero at state **518**, it stays locked. Once the lock count is decremented to zero, it becomes unlocked again at the state **508**.

An unlocked tile is fair game for the tile manager **192** when the memory manager needs to find some space to lock a new tile. Therefore, when the tile manager **192** is looking for space, unlocked memory blocks may be freed and returned to the free memory lists.

The way to go from the unlocked state **508** to the free state **504** is by freeing the handle in which case the memory block is moved onto the free memory list.

Referring now to FIG. **15**, the flow diagram for the InitImageAccess function **412** shows the operation where the tile manager **192** creates the image access context starting at a state **530**. At a state **532** the input parameters are validated. If there is an error with the input parameters, the function ends immediately at an end state **534**.

Input parameters include a document handle indicating which image that the user wants to read or write from. Thus, the document handle must be validated. Another parameter is whether the user wants to read or write to the image. A transformation matrix, also input, basically directs how to scale, rotate, shear, etc., the image data.

If the input parameters are valid, the tile manager **192** locks the document handle at a state **536**. The document handle locks and unlocks just like other structures and resources in the tile manager and it prevents one user of a particular document or image from modifying or deleting that image while another operation or another access context is still using that document.

Then, at a state **538**, the tile manager **192** tests whether a non-orthogonal rotation has been specified. For example, a rotation of 30° causes the tile manager **192** go into a special operation that initializes the access with rotation. That also creates an access context but after a more involved process. Then the tile manager **192** ends the function **412** at a state **534** with a valid access context for rotations.

If an orthogonal rotation is specified then the tile manager **192**, allocates a conventional access context at a state **542**. Then the tile manager **192** continues to a decision state **544** wherein the subimage selection criterion is specified. For instance, the user may request the "low resolution" option which selects the lowest resolution subimage in the document's image stack. (In the context of an image editor, this may be the best solution during zooming or panning.) The user may also specify "most available"—i.e., whatever subimage has tiles currently in cache memory, regardless of the resolution. In either case, the tile manager **192** proceeds to a state **546** to select the reduced resolution subimage that is appropriate to that particular choice, i.e., either the one that has the resolution just greater than what was requested or a subimage whose tiles covering the access region are currently in cache. Now, at a state **548**, the tile manager **192** adjusts the transformation matrix so as to now refer to the reduced resolution subimage rather than the full resolution subimage by adjusting scale factors.

Alternatively, if the state **544** determines that the full resolution subimage is selected then the transformation matrix is unchanged. Proceeding to a state **552**, the pixel and tile limits of the affected image region are calculated. Knowing these limits, in a state **554**, the tile manager **192** creates a temporary directory for the tiles in that region. This directory is a two-dimensional array that references the tiles that contains the affected pixels. Later on the tile manager **192** refers to the region tile directory because it is specific to tiles that are inside the affected region.

The tile manager **192** then initializes the image scaling functions in a state **556**. Such scaling functions presently

used are the subject of applicant's concurrent application entitled "Process for High Speed Rescaling of Binary Images" (U.S. Ser. No. 08/014,085, filed Feb. 4, 1993, which is a continuation of Ser. No. 07/949,761 filed Sep. 23,1992, now abandoned, which is a continuation of Ser. No. 07/693,010 filed Apr. 30, 1991 now abandoned.

Moving on, the tile manager **192** tests whether polygonal clipping is required at a state **558**. For example, a request may be made to only read from within a specific polygonal region. If that is the case, the tile manager **192** initializes the polygonal region clipping functions in the tile manager **192** by passing in the boundary lists. The polygonal clipping function translates the boundary lists into edge lists that are used to very efficiently read out the rows or columns of data.

For example, suppose a "flood" request is made to turn all of the pixels black within an octagonal region. One way to accomplish the operation is to specify the points of the corners of the octagon in image coordinates and pass that in with the initialization of access context request, which would pass those vertices of the polygon into the polygonal clipping function set up function.

Then the tile manager **192** comes to a state **562**, where the tile manager **192** allocates buffers for scaling, if necessary. This is the situation where intermediate copies of the rows or columns of data may need to be kept during the process of scaling. Then the tile manager **192** tests whether the user specified that the region needed to be saved for undoing, at a decision state **564**.

An important feature of the present invention is an "undo" operation that is integrated with the image memory management so that only compressed tiles need to be saved after an undoable edit operation. In this way, a user can easily and quickly retract an edit operation that is no longer desired. For example, in mapping applications, e.g., USGS Quadrangle maps, the impression of a very large map is desired, but it is really composed of smaller map quadrants that were separately scanned, trimmed, adjusted and fit together. The smaller maps can be visually and logically joined into a single, large image. Using the present invention, a user can add a feature, such as a new sub-division, town, or road, that crosses a map boundary, specifying that the feature is undoable. Later, the user can remove the feature modification to the image by specifying the undo operation.

Now at a decision state **568**, the question is whether to update the subimages during the operation. If this is a write operation the tile manager **192** always writes into the full resolution subimage and the changes "trickle down" into the low resolution subimages. But the tile manager **192** has an option as to whether the lower-resolution tiles are updated during the modification operation or later when the tiles are requested for viewing operations. There are advantages in doing them both ways.

For example, if the affected region is small, it is more efficient to update the subimages while progressing through the operation. In this mode, when the tile is unlocked, the manager **192** immediately copies the data down into the next lower subimage tile but only one of the corners of the tile is affected. Thus, only portions of the low resolution subimage tiles need to be modified.

If, however, the subimages are not updated during the operation, then as soon as the image access context is created all of the subimage tiles that overlap the affected region are invalidated (they become "not loaded"). Hence, when the memory manager goes to access them again at some later time, it has to reconstruct them from the higher-resolution tiles. The advantage of that is that the memory requirement at any one moment is half of that of if the tile manager **192** was updating all of the tiles simultaneously. In this way, the tile manager **192** sets a flag at a state **570**.

In state **572** the tile manager **192** "preserves" the affected tiles in the affected subimages. Again, it relates to whether the tile manager **192** is updating subimages or not. If the tile manager **192** is reading, then it preserves only the tiles in the region of the subimage that will be accessed.

The ability to "preserve", or preferentially retain tiles that will be accessed in the course of the operation, is an important feature of the present invention that can yield significantly higher performance in certain situations where memory capacity limitations are encountered. When a tile is "preserved" for a particular access operation, it's preserve count **360** is incremented. The cache manager treats tiles with non-zero preserve counts differently from tiles with zero preserve count. The cache manager will discard unlocked unpreserved tiles before discarding older preserved tiles. (The cache manager normally discards older or less recently used tiles before discarding newer or more recently used tiles.)

Then, within the creation of the access context, the tile manager **192** actually locks down the first row or column of tiles in the region to establish the cache memory requirement for this operation, at a state **574**. If this succeeds, then the caller is assured that there will be sufficient cache space for the entire operation.

The tile manager **192** can perform row or column accesses. However, the following discussion only refers to a row access.

Then, at a decision state **576**, if the tile manager **192** cannot satisfy the request to lock down that first row of tiles, the function **412** terminates at the end state **578**. Otherwise, at state **580** the tile manager **192** initializes the row access functions.

Now, once the tile manager **192** has initialized the row access function in state **580** the tile manager **192** invalidates the affected subimage tiles if the tile manager **192** is writing to the full resolution subimage at a state **582**. Finally, in a state **584** the tile manager **192** returns the handle or a pointer to this access context to the user. From then on the user just uses this pointer to the access context and pointers to input and output buffers to get the next row or column of data.

FIG. **16** illustrates the access context structure **600**. The structure **600** operates on a high level to hide the low level operation from the user and contains bookkeeping information along with some memory management information. The access context **600** contains the following information:

- **602** Pointer to affected doc. Pointer to the document being accessed.
- **604** "Subimage Choice" option value. Specifies how to choose which of the subimages will be read from or written to.
- **606** Index of affected subimage. Index of the specific subimage directly affected by this access context.
- **608** Access quantum. Specifies "granularity" of image access.
- **610** Read/write option. Specifies what type of image memory accesses to prepare for (e.g., read or write).

612 Basic orthogonal rotation value. Specifies the image rotation in terms of how the bits in each buffer row are read from or written to the image (e.g., write buffer row to image column with increasing "y" coordinate).

614 Pixel combination operation. Specifies the pixel operation performed when combining the buffer contents and image contents. The results of the operation are stored in the output buffer when reading. The results go into image memory when writing.

616 Scaler type operation. Specifies the type of scaler preferred. In other embodiments, this may include fast low-accuracy scaling and line width-preserving scaling.

618 "Update overviews" flag. True flag indicates overview subimages should be updated in the course of this modification of the full resolution image. This causes the overviews to be correct when the access is complete.

620 I/O buffer width & height. Width (i.e., row length), total number of rows to process and pitch in pixels of the input/output bitmap.

622 I/O buffer pitch (bytes/row). Pitch of the input/output buffer in bytes used for multi-row accesses. The input/output buffer is assumed to be a contiguous memory bitmap at least as large as the access quanta. It is always read or written in the natural order (by rows, low address to high). Flipping and rotation is always done on the image memory side.

624 I/O buffer bit offset to start of run. Indicates where the buffer's x=0 pixel lies within the first long word of the buffer's storage space. It must be between 0 and 31 inclusive. This parameter allows the caller to match up with arbitrary bit alignments.

626 Rows per strip (for AQ_STRIP access quantum). When operating in the AQ_STRIP mode, this specifies the maximum number of rows per input/output strip. Fewer rows may be written into the last strip if the end of the access region is hit before the strip is filled.

628 Number of I/O buffer rows yet to be processed. This variable is used in the access routines to keep track of the number of input/output rows remaining for the access operation.

630 Pointer to access function used in "SeqBufImageAccess". Pointer to the image access function that is tailored to the specific access mode requested.

632 Stepping directions for image row and column indices. The stepping increment each time the input/output buffer is advanced one row and one pixel. The allowed values are +1, 0, and −1.

634 Pointer to polygon clipping information. Refers to an edge table structure for controlling polygonal boundary clipping.

636 Pointer to raster scaling information. Tile level access information used by lower level modules in the course of the operation.

638 Pointer to uncompressed data in currently locked tiles. Pointer to an array of pointers directly into expanded tile image data. This list is used to accelerate sequential access into image memory. As each new tile row or column is encountered in a sequential access, this array is set to point directly into the affected tiles, which have been brought into cache memory and locked down. In other embodiments this could also be used to point to compressed tiles.

640 Pointer to region tile directory. Pointer to a 2-dimensional array of pointers to the tiles in the affected region of the subimage.

642 Next image row & column to be accessed. The index of the next image row and column to be accessed in sequential row and column operations.

644 Terminal row & column of access region. Stopping values for sequential row and column operations.

646 Unclipped extent of access region. Defines the image region that will be accessed over the course of the operation.

648 Clipped extent of access region. Defines the portion of the requested image region that actually falls within the boundaries of the image. Pixels outside of this rectangle are treated as background pixels.

650 Clipped image buffer bit offset and length. These values specify where, in the intermediate image row or column buffer, the first bit from the clipped image region is located and how many bits are to be read from or written to tiled image memory.

652 Number of tile rows & cols in access region. Number of tile columns and rows in the affected region.

654 Row & column of currently locked tiles. Column and/or row index of the currently locked tile or tiles.

656 Image row & col at origin of first tile in access region. Pixel coordinates of the upper-left pixel in the upper-left tile of the affected region.

658 Number of I/O buffer rows held over for next strip. Number of rows of output data that did not fit into the previous row and must be returned in the next and subsequent rows when expanding while reading image data.

660 Pointer to image tiling/untiling buffer. Points to a temporary buffer to hold data extracted from tiled memory prior to scaling when reading from image memory.

662 Number of bytes in tiling/untiling buffer. Size of buffer in bytes.

664 Bit offset for tiling/untiling buffer. Bit offset to the first valid pixel in tiling/untiling buffer.

666 Access transformation matrix. The transformation matrix mapping input/output buffer pixels onto the pixels of this subimage.

FIG. **17** illustrates the flow diagram for the "Save Region for Undo" function **426** as referenced in FIG. **15**. The tile manager **192** starts at a state **680**, moves to **682** where the tile manager **192** locks the document handle of the affected document that contains the region to save for undo. The tile manager **192** can save multiple regions from multiple documents sequentially and then undo them all in one operation later. Thus, the application programmer is allowed to easily undo multiple-region operations with a single undo call at a later point.

Moving to a state **684**, the tile manager **192** clips the modified region to the image boundaries since there is no information to save outside of the image. Then the tile manager **192** moves to a decision state **686** wherein the tile manager **192** tests whether the affected region overlaps the image. If there is no overlap, that is to say, there is no image data to save, then the tile manager **192** moves to a state **688** where the tile manager **192** unlocks

the document handle and terminates the function 426 at an end state 690.

If, however at state 686, the modified region does overlap the image, the tile manager 192 moves to a state 692 wherein the tile manager 192 allocate memory for an "undo region header". The undo region header is similar to a document header, but reduced comparatively in the amount of data conveyed therein. The undo region header will be associated with tile header information, etc.

The tile manager 192 then moves to a state 694 where the tile manager 192 allocates memory for "undo region tile headers". These tile headers will be used to store copies of the original versions of the tiles in the affected region. The tile manager 192 then proceeds to a state 696 wherein the tile manager 192 makes an "undo tile directory".

Then the tile manager 192 moves to a loop state 698 where the tile manager 192 loops for each tile row in the region. The tile manager 192 then transitions to a loop state 700 wherein the tile manager 192 loops again for each tile column in the region (Thus, there is a two-dimensional loop.)

The tile manager 192 moves from the state 700 to a decision state 702 where the tile manager 192 checks to see if that particular tile in the document is loaded in the image cache memory. If the tile is not loaded, the tile manager 192 skips to the next tile in the region by returning to the loop state 700. OtherWise, if the tile is loaded, the tile manager 192 marks the undo copy of the tile as loaded in a state 704.

Note that there are two tiles. One is the original version of the tile that is still associated with the document and the second is the copy that the tile manager 192 is going to make and associate with the undo region header.

At a decision state 706, a test determines whether the document tile is blank. If the tile is blank (i.e., all background color), then the tile manager 192 moves to a state 708 and simply marks the undo tile as "blank" and returns to the FOR-loop at 700. If the document tile is not blank, then the tile manager 192 moves to a state 710 and the tile manager 192 marks the undo tile as "not blank" and moves to a state 712 Wherein the tile manager 192 tests whether the document tile has a valid copy of compressed data on the disk.

If a valid copy of compressed data does reside on disk, the tile manager 192 moves to a state 714 and simply copies the compressed tile disk location and size information from the document tile header to the undo tile header. Note that it is possible for a particular tile to have multiple representations of the same data. That is, a compressed version and an expanded version of the tile may exist in cache simultaneously. And a tile may have a compressed version in cache as well as on the disk. For undo, the strategy is to store the most compact version possible. The most compact version with regard to cache memory usage is to have a copy of the compressed tile on the disk.

If there is no compressed copy of the tile on the disk, the tile manager 192 proceeds to a decision state 716 wherein the tile manager 192 determines whether an uncompressed copy of the document tile resides on the disk. If the test succeeds, the tile manager 192 enters a state 718 and copies the uncompressed tile disk location and size information from the document tile to the undo tile and then returns to the inner FOR-loop at a loop state 700.

If, at state 716, there is no uncompressed tile information on the disk, the tile manager 192 continues execution to a state 720 in FIG. 17B wherein the tile manager 192 locks the compressed version of the document tile. This locking of the compressed version of the document tile may cause an expanded version of the document tile to be compressed and a compressed version created. Therefore, there is a possibility of an error and that is checked at the decision state 722.

If there is an error than the tile manager 192 unlocks the document handle at a state 724 and terminates with an error condition at the end state 726. If there was no error in locking the compressed version of the tile then the tile manager 192 moves from the state 722 to a state 728 wherein the tile manager 192 allocates and locks down cache memory for a copy of the compressed data to be associated with the undo header. There is another error possibility at this point and the tile manager 192 checks for an error at a decision state 730. If there is an error then the tile manager 192 returns to a state 724 and thereafter terminates the function 426.

If there was no error in locking cache memory at the state 730, the tile manager 192 moves to a state 732 and copies the compressed data from the document tile to the undo tile. The tile manager 192 actually copies the data that is stored within the tile—i.e., the compressed image data is copied from the document version to the undo version. Then the tile manager 192 moves to a state 734 and unlocks the compressed version of the document tile. Now, at a state 736, the tile manager 192 unlocks the compressed version of the undo tile and the tile manager 192 returns to the inner FOR-loop at state 700 on FIG. 17A where the tile manager 192 loops back to continue the loop for all of the tiles in the affected region.

When the tile manager 192 is done with all of the tiles in the affected region, the tile manager 192 moves to a state 738 where the tile manager 192 links the new undo header into the undo region list. Thus, multiple regions can be saved in the undo list and then in one operation, by calling undo previous raster operation, all of the operations that had been accumulated, can be undone. Then the tile manager 192 moves to a state 742 wherein the tile manager 192 unlocks the document handle and terminates the function 426 normally.

FIG. 18 shows the load tile to raster image function (LoadTiff). FIG. 18 is a flow diagram for the part of LoadTiff that loads tiled images only. In reference to FIG. 18, the overall process may be understood whereby an existing file on the disk, i.e., an image file on disk, is mapped into memory. As described below, the overall process permits loading large images in a short time period relative to how long it would take to actually copy all of the image data into the computer's memory. In accordance with the present invention, the process shown in FIG. 18 is called the indirect loading capability. As shown in FIG. 18, the tile manager 192 begins the LoadTIFF function 408 at a start state 750 and moves to a state 752 where the tile manager 192 opens the input file that is on the disk. If there is an error on the disk, the tile manager 192 prints an error message at a state 754 and terminates at an end state 756. If no error exists, then the tile manager 192 moves to a state 758 and checks for the TIFF header structure that identifies that the input file is in fact a TIFF file. While the disclosure below discusses a TIFF file, it is to be understood that the process shown in FIG. 18 may be per-

formed on all types of tiled files, such as a MIL-R-28002A Type II file or an IBM IOCA tiled file.

Still referring to FIG. **18**, if the tile manager **192** finds something other than TIFF header structure at state **758**, the tile manager **192** moves to state **754** to indicate an error, and then exits at the end state **756**. If the tile manager **192** finds a TIFF header structure while at state **758**, the tile manager **192** move to a state **760**, wherein the tile manager **192** counts the number of subimages in the TIFF file, one or more of which may exist in a TIFF file.

Next, the tile manager **192** moves to a state **762** and reads the full resolution subimage information which constitutes the basic information about the image, e.g., the image width and height, the size of the tiles, the compression format that is used, and the resolution. If the basic image information is not present and in proper form, the tile manager **192** moves to the state **754** to indicate an error. On the other hand, if no error is indicated at state **762**, the tile manager **192** moves to state **764**, wherein the tile manager **192** creates a skeleton document and locks that document. The skeleton document at this point contains no cache memory but only tile directory and tile headers that represent in a virtual sense the tiles that compose the image.

The tile manager **192** next moves to a state **766** where the TIFF full resolution subimage tile information is loaded into the tile headers for the full resolution subimage, as more fully disclosed below in reference to FIG. **19**. Next, the tile manager **192** moves to a loop state **768** where there is a loop for each of the remaining lower resolution subimages. While in this loop, the tile manager **192** accesses a decision state **770**, wherein the tile manager **192** determines whether

$$fr/lr=2^n \quad (1)$$

where

fr is the full resolution subimage resolution in pixels per inch; and

lr is the particular low resolution subimage resolution in pixels per inch.

If the ratio of fr to lr is a power of two, then a successful test is indicated, and the tile manager **192** moves to a function **424** and loads the TIFF subimage tile information into the tile headers for that particular subimage level. On the other hand, if the ratio of fr to lr is not a power of two, as indicated at the decision state **770**, then the tile manager **192** ignores the particular subimage under test and returns to the state **768** until all of the subimages in the file are processed. When all subimages have been processed, the tile manager **192** moves to a state **772** and unlocks the document handle of the newly created document and terminates normally at an end state **756**.

Now referring to FIG. **19**, the function **424** whereby the tile manager **192** loads the TIFF subimage tile information into tile headers is shown. More particularly, the tile manager **192** begins at a start state **780** and moves to a state **782** wherein the tile manager **192** reads the number of tiles in the subimage. Then the tile manager **192** moves to a state **784** wherein the tile manager **192** allocates temporary buffers for the tile mode offset and byte count lists. These three lists have one entry each per tile in the subimage. If the tile manager **192** cannot properly allocate the temporary buffers, then the tile manager **192** exits with an error condition at an end state **786**.

Upon successful allocation of the buffers, the tile manager **192** moves to a state **788** where the tile manager **192** reads the tile offset and byte count information from the disk file into the allocated buffers. In the TIFF file standard, all tiles are stored in the same mode (e.g., compressed). However, other tiled file formats (e.g., MIL-R-28002A Type II) specify the storage mode for each tile. The tile mode simply states whether a particular tile is stored in compressed form, in uncompressed form, or whether the tile is all foreground or background color. The tile manager **192** next moves to a state **790** where the tile manager **192** fills in the tile storage mode list. At state **790**, the tile manager **192** synthesizes the tile mode information that the TIFF file does not contain itself. Then the tile manager **192** moves to the function **425** wherein the tile manager **192** stores the information in the subimage tile headers (FIG. **10**), and terminates at an end state **786**.

Now referring to FIG. **20**, the function **425** whereby the tile manager **192** stores file information in tile headers is shown. The tile manager **192** begins this process at a start state **800** and moves to a state **802** where the tile manager **192** locks the document handle of the document for which the tile manager **192** is loading the subimage for. This function is performed once per subimage in the file and there may be multiple subimages in the file. Consequently, the locking of the document handle function can be performed several times in the process of loading a single document.

As shown in FIG. **20**, in the event that an error occurs in locking the document handle the tile manager **192** terminates at an end state **804**. On the other hand, if the tile manager **192** successfully locks the document handle at state **802**, the tile manager **192** moves to a state **806** where the tile manager **192** determines whether the number of tiles in the file matches the number of tiles expected for the particular subimage in the particular file or document. If a mismatch exists between the actual and expected number of tiles, the tile manager **192** moves to a state **808** to print an error message and then terminates at the end state **804**. On the other hand, in the event that the number of actual tiles matches the number of expected tiles, the tile manager **192** moves to a loop state **810** where the tile manager **192** enters the first part of a FOR-loop for each tile row. Still referring to FIG. **20**, the tile manager **192** moves from state **810** to state **812** for each tile column. Accordingly, it will be understood that the tile manager **192** is processing a two-dimensional array at the states **810**, **812**.

In accordance with the present invention, the tile manager **192** processes, at states **810**, **812**, all of the tiles required to cover the particular subimage. Next, the tile manager **192** moves to a decision state **814** wherein the tile manager checks the value in the tile mode entry to determine whether the tile data is compressed. If the tile data is compressed, the tile manager **192** moves to a state **816** and stores the file offset and byte count in the compressed tile handle. The compressed tile handle is a part of the tile header structure, and the file offset is the location of the compressed data for the particular tile within the file as measured by a byte offset from the start of the file. The byte count represents the number of bytes of compressed data associated with the particular tile starting at the offset that is provided at the tile. From state **816**, the tile manager moves to state **828**, wherein the tile manager sets a flag to indicate that the particular tile is not blank.

In the event that the tile manager determines at state **814** that the tile data is not compressed, the tile manager

192 moves to a decision state **818** where the tile manager **192** checks to see if the data is uncompressed. If the data is uncompressed on the disk, the tile manager **192** stores the file offset byte count information in the uncompressed tile handle in state **820**. From state **818**, the tile manager moves to state **828**, wherein the tile manager sets a flag to indicate that the particular tile is not blank.

If the tile manager **192** determines at state **818** that the tile data is not uncompressed, then the tile manager **192** moves to state **822**, wherein the tile manager **192** checks to see whether the tile is all foreground at a state **822**. For example, in a black and white drawing engineering document, foreground color is black, so the tile manager **192** treats a foreground as a black tile. If the tile is determined to be a foreground tile, the tile manager **192** proceeds to state **824**, wherein the tile manager **192** creates an all foreground tile, and then sets the flag as not blank at state **828**. As an example, if the image being processed is a color image, the tile manager **192** could fill the tile with the foreground color at the state **824**.

On the other hand, if the tile is not all foreground, the tile manager proceeds to state **826** to determine whether the tile is all background. As discussed above, binary images usually have background pixels which are white or zero value. If a particular tile is blank, the tile manager **192** moves to a state **828** where the tile manager **192** sets the blank flag to indicate that the tile is indeed a blank tile. If at the state **826** the tile manager **192** determines that the tile is not all background, the tile manager **192** terminates with an error at an end state **830**. In other words, having determined at state **822** that the particular tile was not all foreground, the only possibility left at state **826** is that the tile is all background. Consequently, a determination at state **826** that the tile is not all background indicates an error.

From state **828**, the tile manager **192** moves to a state **832** and sets the loaded flag to true indicating that a valid image information set has been associated with the particular tile. The tile manager **192** completes the loop described above for each tile. After having processed each tile in the particular image, the tile manager **192** exits the two FOR-loops and moves to a state **834** where the tile manager **192** unlocks the document handle and then terminates normally at the end state **830**.

Now referring to FIG. **21**, the tile manager **192** performs a function which for purposes of the present invention will be termed "Undoable Raster Operation". The function shown in FIG. **21** is performed by the tile master **192** in the function "Begin Undoable Ras-Op", and is a relatively simple function, the purpose of which is to clear the undo region list. More particularly, in the process shown in FIG. **21**, the tile manager **192** frees all of the undo regions associated with the previous operation to prepare for a new undo operation. Indeed, the present invention could be configured to have multiple level undo, i.e., the system of the present invention could undo two or three or more operations going into the past and also to be able to redo all of those operations at the user's choice. For example, the last three operations could be undone and then the oldest of those operations redone.

In specific reference to FIG. **21**, the tile manager **192** begins at a start state **840** and then proceeds to loop state **842**, in which the tile manager **192** executes a FOR-loop for each undo region in the current list. The tile manager **192** loops to a state **844** where the tile manager **192** frees all of the memory associated with that undo region. This may include freeing compressed data that is stored in cache or expanded data that is stored in cache and associated with the undo region. When the tile manager **192** finishes all of the regions, the tile manager **192** terminates at an end state **846**.

Now referring to FIGS. **22A** and **22B**, there is shown the control flow for the ReadRowToRow function **414** which produces one or more rows of scaled image data each time it is performed. It is one of the basic image access functions. It should be understood that the tile manager **192** can also read columns of an image, etc., so as to produce a rotated output.

The tile manager **192** enters the function **414** by moving to a start state **850** and proceeds to a decision state **852** where the tile manager **192** checks for a region overrun. In other words, when the access context is created, the region that is going to be read in the course of the overall operation is specified, and in the event that the read row to row subfunction is accessed too many times, the region will be overrun. Any such overrun is detected by the tile manager **192** at state **852** and reported at state **854**. In the event of an overrun, the tile manager **192** terminates at an end state **856**.

If, on the other hand, no region overrun has occurred, the tile manager **192** moves to a decision state **858** where the tile manager **192** checks to see whether old results are carried over to the new strip. Such a carryover could occur when, for example, raster data is being enlarged by expanding one or more lines from the image. For example, when raster data is being enlarged by 4×, each line of input generates four (4) lines of output. Accordingly, three (3) output rows could be carried over for later strips. With this eventuality in mind, the tile manager **192** ascertains whether any data is being carried over and if so, the tile manager **192** uses the carried-over data before generating a new row. Consequently, if there is new data carried over, the tile manager **192** moves to a state **860** where new rows are generated from the carried over data.

Next, the tile manager **192** moves to a state **862** where the tile manager **192** checks to see if a particular strip is full. For purposes of the present invention, a strip is a collection of rows, i.e., a set of numbers arranged in rows As indicated at state **862**, if the strip is full, then the tile manager **192** ends at the end state **856**.

If the strip is not full and the tile manager **192** has used up all the carried over data, then the tile manager **192** moves to a decision state **864** where the tile manager **192** checks for ghosting, i.e., the skipping of some rows of data in order to produce a low quality image while panning or zooming. If ghosting is in effect, the tile manager **192** moves to state **866**, wherein the tile manager **192** calculates the number of blank lines to create. The system then moves to a state **868** where the tile manager **192** writes the blank lines to the output strip buffer.

From state **864**, if no ghosting was detected, or state **868**, if ghosting is not in effect, the system moves to state **870** where the tile manager **192** again checks to see if the strip buffer is full. If it is, the tile manager **192** exits at the end state **856**. If it is not, the tile manager **192** checks to see that there are still input rows to read in a decision state **872**. If there aren't, the tile manager **192** has reached the end of the specified image region, and proceeds to state **874** to obtain another row of output data by flushing the scaler buffers. In accordance with the present invention, in the state **874** the tile manager **192** sets a flag that is subsequently passed down to the

scaler functions to flush intermediate results from the scaler functions. This is the case when for reducing data, i.e., if a plurality of rows is being combined into one output row. That is how the last output row is produced.

From state **874**, the system moves to state **894**, shown in FIG. 22B. On the other hand, in the event that there are no unread image rows at state **872**, the system moves to decision state **876**, where the system determines whether the row is outside of the valid image boundaries. If yes, the system moves to a state **878**, where the tile manager **192** substitutes blank lines for the input. The tile manager proceeds from state **878** to a state **894**, shown in FIG. 22B. If the answer to the decision at state **876** is no, the system moves to a decision state **880**, shown in FIG. 22B, to check whether the row is contained in the currently locked tile row.

At state **880**, the tile manager **192** moves down the image, and the system sequentially passes through successive tile rows. Each tile contains, e.g., **512** rows, so when a particular tile row is locked it stays locked until all **512** image rows in that tile row have been read. Each time the system arrives at a new row it tests to see that the row is contained in the currently locked tile row. If it is not, the system moves to the state **430** (function ExpTileUnlock) to unlock the old tile row and lock down the new tile row (at state **428**). In addition, the tile manager **192** has to unpreserve the row of tiles that was just unlocked. Unpreserving them tells the memory manager that those tiles are no longer needed for this access operation and it can do what it wishes with them.

Next, the system proceeds to a decision state **882** to determine whether any tiles are blank. If they are, the tile manager **192** substitutes a reference to a "common blank tile" and that common blank tile is used, as indicated at state **884**. All tiles that are blank are mapped onto this common blank tile. Consequently, the tile manager **192** uses less image memory.

From state **884**, **882**, or **880**, as appropriate, tile manager **192** proceeds to a decision state **886** to check for polygonal clipping. If the tile manager **192** is doing polygonal clipping then each input row of data is clipped as appropriate for that polygon in states **888** and **890**. The loop allows multiple clipped regions within each row. If there is no clipping, then the tile manager **192** simply copies the entire input row from the image into the input row buffer in a state **892**. Then the tile manager **192** move to a state **894** where the tile manager **192** passes these input rows through the scaler if the tile manager **192** is scaling the data. Finally, the tile manager **192** takes the results of the scalers and copies that information to the output strip buffer if necessary at a state **896**. The tile manager **192** then returns to the state **870** (shown in FIG. 22A) where the tile manager **192** continues the process of retrieving input rows and scaling them until the tile manager **192** has filled the output strip buffer. The system then moves to the termination condition at the end state **856**.

Now referring to FIG. 23A, a process which will be referred to as "Write Rows to Region" will be described. The tile manager **192** starts at state **900** and moves to state **902** where the tile manager **192** tests for region overrun. Region overrun can occur when the calling function attempts to write more rows to the image than was specified when the access context was created. If the region was overrun, the tile manager **192** reports an error at state **904** and terminates with an error at state **906**. If there is no region overrun, the tile manager **192** moves to the FOR-loop in state **908** where the tile manager **192** loops for each input row in the input buffer, which is the buffer that is passed in by the calling function. It contains the data that is to be processed and written to the image. The loop is executed for each row and moves to state **910** where the input data is passed through the scaler functions and put into a temporary buffer. If the scaler does not always produce an output row, as is the case when reducing the resolution, a plurality of input rows may have to be combined to produce a single output row. So, at the state **912**, the tile manager **192** determines whether an output row was produced after the input row is scaled. If not, the tile manager **192** goes back to the loop at state **908** and continues the process as described. On the other hand, when the tile manager determines at state **912** that an output row was produced, the tile manager **192** moves to state **914** which is a FOR-loop for each copy of the scale row to write to the image. It may be the case that more than one copy of the scaled row needs to be written into image memory. This is the case when the tile manager **192** is expanding the input image data. It may be that one input row is replicated four times to get a 4× expansion factor.

Next, the tile manager **192** moves to state **916** where the tile manager **192** checks to see if the destination row index is outside of the image's clipping boundaries. If so, the tile manager **192** simply ignores it and moves back to state **914**. If it is within the clip boundaries the tile manager **192** moves to state **918** where the tile manager **192** determines whether the destination row is in the currently locked tile row. If it is not, the tile manager **192** moves to state **920** where the tile manager **192** unpreserves and unlocks the old tile row that is currently locked. The tile manager **192** then moves to state **922** to determine whether the update overview flag is true. This is an option that is specified in the lo access context and it determines how lower-resolution tiles are updated when the full resolution subimage is modified. If the update overview flag is true, then the tile manager **192** moves to state **924** where the tile manager **192** unpreserves the low resolution tiles that will no longer be needed.

After the system has unpreserved the low resolution tiles that are no longer needed at state **924**, the system moves to state **926** and locks down the new tile row. Only the full resolution tile row is locked at this level. The low resolution tiles are actually updated when the call to unlock the old tile row is made.

Next, the tile manager **192** moves to state **928** to determine whether an error was detected when the new tile row was locked. If so, the system terminates with an error condition at state **906**. If there is no error or if in state **918** the tile manager **192** finds that the destination row is currently in the locked tile row, the tile manager **192** moves to state **930** in FIG. 23B. At state **930**, the tile manager **192** determines whether polygonal clipping is activated. If it is, the tile manager **192** computes the clip points for the current image row, as indicated at state **932**, which results in a list of clip point pairs.

The tile manager **192** then moves to state **934**, wherein the tile manager **192** conducts a FOR-loop for each of the clip point pairs that the tile manager **192** computed in state **930**. As shown in FIG. 23B, the tile manager **192** loops to state **936** where the tile manager **192** copies pixels from a scaler output buffer to the image row between each pair of clip points. When that loop terminates, the tile manager **192** returns to state

914 in FIG. 22A. On the other hand, if the tile manager determines at state 930 that polygonal clipping is not active, the tile manager 192 moves to state 938, wherein the tile manager 192 copies the scaler output buffer pixels to the image row without clipping. The tile manager 192 then proceeds to state 914.

Now referring to FIG. 24, the tile manager starts at state 950 in the end access function shown in FIG. 24 and proceeds to state 952. At state 952, the system cleans up after row or column access functions by freeing buffers used by the row or column access functions.

Next, at state 954, the tile manager 192 unlocks the last row or column of tiles accessed. Then, the system moves to state 956 where the tile manager 192 unpreserves any tiles in the region that are still preserved. The system may perform the functions at states 954, 956 when an operation was aborted in mid-progress and it cleans up after those partially completed operations.

At state 958, the tile manager 192 cleans up after the polygonal clipping function. If there was polygonal clipping involved in this access context the tile manager 192 has to free the buffers that contain the polygon edge information.

Next, the system moves to state 960, where the tile manager 192 frees scaler buffers, the temporary tile directory, etc.. From state 960, the system moves to state 962, wherein the tile manager 192 unlocks the document handle to indicate to the memory manager that the access context no longer is referring to the particular document associated with the document handle.

The tile manager 192 next moves to state 964 where the memory that was used to store the data for the access context is freed. Then, the system ends the clean up function at state 966.

Referring now to FIGS. 25A,B, a function is shown which, for purposes of the present invention, will be termed the "Undo Previous Raster Operations". The tile manager 192 starts at state 970 and moves to state 972, wherein the tile manager determines whether any undo regions exist in the list or if the list is empty. If no regions exist then the tile manager 192 moves to end state 974 and terminates normally.

If the tile manager 192 determines at state 972 that "undo" regions do exist, the tile manager 192 moves to state 976, where the tile manager 192 enters a loop for each undo region in the list. In this loop, the tile manager 192 moves to state 978 where the tile manager 192 locks the affected document handle. The document handle that is locked is the one that was stored in the undo region header that tells where that particular undo region came from. The tile manager 192 moves from state 978 to state 980 where the tile manager 192 saves the current document region to support redo (i.e. an "undo" operation following by another "undo" operation). Then the tile manager 192 moves to state 982 to invalidate the affected tiles in the lower-resolution subimages. The strategy represented by states 980, 982 in FIG. 25A is to save the minimum amount of information that is needed to reconstruct the image, which means the tile manager 192 saves only the affected tiles in the full res subimage.

Next, the system moves to a loop indicated by the states 984, 986. In this loop, for each tile, the tile manager 192 moves to state 988, discarding the document tile image data. Then the tile manager 192 moves to state 990 to determine whether the undo tile is loaded. If it is not loaded, the tile manager 192 moves to state 992 where the tile manager 192 marks the document tile as "not loaded". If the tile is determined to be loaded at state 990, the tile manager 192 moves to state 994 to mark the document tile as "loaded". From state 994, the system moves to state 996 in FIG. 25B.

At state 996, shown in FIG. 25B, the tile manager 192 determines whether the undo tile is marked as blank. If it is, the tile manager 192 moves to state 998, wherein the tile manager marks the document tile as blank, and then the system loops back to state 986. If the undo tile is determined to be not blank at state 996, the tile manager 192 move to state 1000. At state 1000, the tile manager 192 checks to see if the undo tile points to compressed data on the disk. If it does, the tile manager 192 moves to state 1002 and copies the disk location and size information about the compressed data into the document tile header and loops back around. If there is no compressed data on the disk, then the tile manager 192 moves from state 1000 to state 1004, wherein the tile manager 192 determines whether uncompressed data exists on the disk associated with the undo tile.

If so,the tile manager 192 moves to state 1006, wherein the file manager 192 copies the disk location and size information about the uncompressed data into the document tile header and loops back to state 986. If the system determines at state 1004 that there is no uncompressed data on the disk, the tile manager 192 proceeds to state 1008, wherein the tile manager 192 determines whether the undo tile "points" to uncompressed data in cache memory. If it does, the tile manager 192 moves to state 1010, wherein the tile manager 192 copies the pointer to the uncompressed data from the undo header to the document tile header.

From state 1010, the system returns to state 986. If no uncompressed data exists in the cache, however, as determined in state 1008, the tile manager 192 stores a pointer to the compressed data in cache in the document tile header and returns to state 986.

Referring back to FIG. 25A, when the tile manager 192 has completed the loop described above, the system moves to state 1014, unlocking the document handle. From state 1014, the tile manager 192 proceeds to state 1016, wherein the tile manager 192 frees the memory associated with the undo header. The tile manager 192 then moves to state 976. Thus, the system returns to state 976 for each undo region in the list. As intended by the present invention, the tile manager 192 continues the loop for all of the regions in the list. The undo regions are restored in "last-in-first-out" order. At the completion of the looping process described above, the system moves to state 974.

Now referring to FIG. 26, when the tile manager 192 ends the cache management, the tile manager 192 starts the process shown in FIG. 26 at state 1020 and proceeds to state 1022 wherein the system frees the compression buffer. From state 1022, the system proceeds to state 1024, wherein the system frees the common blank tile. Next, the system moves to state 1026 to free the tile cache memory. The system then ends the process shown in FIG. 26 at state 1028.

FIG. 27 provides an explanation of the function exp tile lock. The tile manager 192 starts at state 1040 and moves to state 1042 where the tile manager 192 enters a FOR-loop for each tile row to be locked. In accordance with the present invention, the system in the exp tile lock function is capable of locking down all the tiles in a two dimensional region.

For each tile in the specified region, the system moves to state **1046**, wherein the tile manager **192** determines whether the particular tile is blank. To make this determination, the system examines flags in the tile header itself or checks the image data for that tile to determine if there are any non-background pixels. If it is not a blank tile, the tile manager **192** move to state **434** where the tile manager **192** locks the uncompressed version of the tile. Then the tile manager **192** proceeds to state **1050**, wherein the tile manager **192** determines whether an error had occurred in the process of creating the uncompressed version of the tile. If no error is found at state **1050**, the tile manager **192** continues to loop to the next tile in the region by returning to state **1044**. If an error did occur, as determined at state **1050**, the system proceeds to state **430** to unlock previously locked tiles, and then ends at state **1056**.

In the event that the tile manager **192** at state **1046** detected that the particular tile was a virtual blank tile, i.e., a tile that exists only by virtue of the fact that there is a tile directory entry for that tile, the tile manager **192** take no action, other than to loop back to state **1044** for further processing.

FIG. **28** illustrates the control flow for the "lock expanded tile" function **434** wherein the tile manager **192** takes a single tile and locks the expanded version of the tile in the image data cache **194**. The tile manager **192** enters the function **434** at a start state **1060**, and proceeds to a decision state **1062** wherein the tile manager **192** tests whether the tile is marked as "loaded". As already mentioned, a loaded tile is one that either contains or references valid image data, is either uncompressed or compressed image data, and it either resides in cache memory or on the disk. If the tile is not loaded, the tile manager **192** moves to a function **436** wherein the tile must be created from higher resolution tiles which are loaded. Afterwards, the tile manager **192** determines if there was an error in a decision state **1066**. If there was an error, the tile manager **192** terminates the function **434** at an end state **1068** and reports the error condition. Otherwise, if there was no error in creating the tile, the tile manager **192** continues, moving from the state **1066** to a decision state **1070**.

The tile to be locked is now loaded so the tile manager **192** tests whether the uncompressed version of the tile is in cache memory. The objective of the function **434** is to guarantee that there is an uncompressed version of the tile in cache memory. Now, if the uncompressed version is not in the cache, the tile manager **192** proceeds to a decision state **1072** to determine whether the selected tile is a blank tile.

If the tile is blank, the tile manager **192** proceeds to a state **438** to create a blank tile. Note here that the function ExpTileLock **428** (FIG. **27**) will detect a blank tile before calling the function **434** if it can take advantage of using a common blank tile at a higher level. In other words, if the tiles are locked for reading only, i.e., the image data will not be modified in any way, then all blank tiles can refer to the same section of blank memory. However, if the tiles are locked for writing, all tiles must have their own memory because different image data can be written to the different tiles.

At this point, state **438**, memory has presumably been allocated for a blank tile. Moving to a state **1074**, the tile manager **192** tests whether there was an error and moves to the end state **1068** if there was an error.

Returning in the discussion to the decision state **1072**, if the tile is not blank, then the tile manager **192** transitions to a decision state **1076** and tests whether there is a uncompressed version of that tile on the disk. If the uncompressed version is on disk, then the tile manager **192** reads that uncompressed version from the disk into cache memory at a state **1078**. Then the tile manager **192** moves to the state **1074** to test for errors.

If, at the state **1076**, there is not an uncompressed version on the disk, the tile manager **192** moves to the function **440** so as to create the tile from the compressed version. The compressed version can be either in cache memory or on the disk, and this is handled by the function **440**. Again, the tile manager **192** checks for an error at the state **1074**.

Now, assuming that there was no error found at the state **1074**, the result is that the tile manager **192** has an uncompressed version of the tile in cache. Therefore, the tile manager **192** proceeds to a decision state **1080** to verify that the uncompressed version is valid. It is sometimes the case that the uncompressed version of a tile is locked by one access context and then for come reason it is invalidated by another access context. This happens when the first access context is reading an uncompressed version of a tile from a lower resolution image, and another access context is actively modifying the full resolution subimage with a particular setting of parameters. If the tile not valid, the function **434** is terminated at the end state **1068**.

Alternatively, a valid tile that was determined at the state **1080** causes the tile manager **192** to increment the uncompressed data lock count for that tile at a state **1082**. The lock count starts out at zero for an unlocked tile and can increment as high as necessary. However, the lock count will be decremented once for each unlocking operation. It is important to match the number of times a tile is locked with the number of times the tile is unlocked. Otherwise, the tile would end up in a permanently allocated (unfreeable), locked state.

Proceeding to a decision state **1084**, the tile manager **192** tests whether the tile is locked for writing or for reading. If the tile manager **192** locked the tile for writing, the execution of the function **434** continues to a state **1086** wherein the "blank" status flag is invalidated. The blank status flag is actually a combination of two flags. One that says that the tile is blank or not blank and the second flag that says if the first flag is valid or not. The reason for two flags is that the way to detect that a tile is blank is by searching through all the pixels in that tile. To do so every time the file is accessed would be wasteful so occasionally, truly blank tiles won't be handled as blank tiles. Hence, there is a second flag that is set, in the state **1086**, when the first flag is invalid. The second flag indicates that the tile must later be examined to determine whether it is still blank.

The tile manager **192** next moves to a state **1088** to invalidate the disk-resident, uncompressed version of the tile, if one exists. This is because the tile manager **192** will modify the cache-resident version of the tile. To synchronize the cache-resident and disk-resident versions, the disk-resident version is invalidated. Then, at a state **1090**, the tile manager **192** invalidates and frees the compressed versions if they exist.

A compressed version of the tile may be in cache or on the disk and, at the state **1090**, the tile manager **192** cleans both out of memory. Thus, at the end of the "lock for writing" operation, the only valid version of the tile is the expanded version in cache, which at this point is locked. Then the tile manager **192** continues to a state **1092** to move the newly locked, expanded ver-

sion of the tile to the front of the "most recently used (MRU)" list of uncompressed tiles.

The MRU list is a doubly-linked list wherein, starting at the beginning, the tile is found that was most recently used, then the next most recently used, and so on, the last tile was used the longest time ago. That list is used by the cache manager to determine which tiles are least likely to be used again as a second level of criteria.

Finally, the tile manager **192** terminates the LockExpHandle at the end state **1068**.

FIG. **29** illustrates the control flow for the "unlocking expanded image tile group" function **430**. The function **430** is just the reverse of lock expanded image tile group. In other words, there is a region of locked tiles which must be unlocked because the access to the tiles is complete. Generally, the two functions, ExpTileLock and ExpTileUnlock are called for a row or column of image data rather than a region but an entire region lock/unlock is possible.

The tile manager **192** enters the function **430** at a start state **1110**. The loop states **1102** and **1104** represent the beginning of nested FOR-loops. That is, the outer loop, beginning at the state **1102**, unlocks a row of tiles, and the inner loop, beginning at the state **1104** unlocks a column of tiles. Moving from the state **1102**, to the state **1104**, and then to the function **432**, the tile manager **192** unlocks the uncompressed version of the tile. When all the tiles in the region are unlocked, the tile manager **192** terminates the function **430** at an end state **1108**.

Now referring to FIG. **30**, the tile manager **192** enters the UnlockExpHandle function **432**, referred to in FIG. **29**, at a start state **1110**. The tile manager **192** proceeds to a decision state **1112** to test whether the uncompressed version of the currently selected tile is in fact locked, i.e., whether the lock count is non-zero. If the tile is not locked, the tile manager **192** exits the function **432** at an end state **1114**.

If, at the state **1112**, the tile is found to be locked, the tile manager **192** moves to a state **1116** to decrement the lock count. Thereafter, the execution continues to a decision state **1118** wherein the tile manager **192** tests whether the "update overview" flag is set true. If the flag is set, the tile manager **192** moves to a state **1120** to update the corresponding lower-resolution tiles. In the process of modifying tiles, the tile manager **192** locks a tile down in the image data cache to write to it. When the tile is unlocked, that is a signal to the memory manager to update the lower resolution tiles that correspond to the higher resolution tile. Thus, the image data in the high resolution tile being unlocked is copied down into the lower resolution tiles, all the way down to the bottom of the image stack.

Once the lower resolution images are modified, or if the overviews are not being updated, the tile manager **192** proceeds to a decision state **1122** to test whether the lock count is exactly zero. If the lock count is not zero, the tile manager **192** terminates the function **432** at the end state **1114**.

Otherwise, the tile manager **192** moves to a state **1124** to clear the "cache collection delay" flag. The cache collection delay flag is set by the tile manager after unsuccessfully trying to reduce the expanded memory usage of the cache file. It is cleared in the function **432** because there is now the possibility of freeing the tile that was just unlocked. In other words, the tile can be removed from the cache to create some space. This flag prevents the tile manager or the cache manager from making repeated, unsuccessful attempts to create space.

After the tile manager **192** clears the flag, execution proceeds to a decision state **1126** to determine whether the uncompressed version of the tile is invalid. As explained hereinabove, it is possible for one access context to have the expanded version of the tile locked down and another access context to invalidate the data in that tile. The tile must remain in memory until the first access context unlocks the tile. Once it is unlocked and the lock count is decremented to zero, if the tile is invalid, the tile manager **192** moves to a state **1128** to free the uncompressed tile version, or remove the tile from the image data cache. In either case, the tile manager **192** terminates the function **432** at the end state **1114**.

FIG. **31** illustrates the control flow for the "create tile from higher-resolution tiles" function **436** referred to in FIG. **28**. The tile manager **192** begins the function **436** at a start state **1140** and proceeds to a decision state **1142** to determine whether the tile is in fact already loaded, in which case no further processing is needed and the tile manager **192** terminates the function **436** at an end state **1144**. Assuming that the tile is not loaded, the tile manager **192** moves to a decision state **1146** to test whether a higher resolution subimage exists.

This function is called only for lower resolution subimages where the tile manager **192** can create the lower-resolution tiles from higher-resolution tiles. Hence, higher-resolution subimages must exist for the function to succeed. If no higher-resolution subimages exist, the tile manager **192** reports the error and terminates the function **436** at the end state **1144**.

If the higher-resolution subimage does exist, the tile manager **192** proceeds to a state **1150** to calculate the indices of, or locate, the four higher-resolution tiles that reduce to this tile. There are four tiles involved because the preferred resolution step between subimage levels is two in the presently preferred embodiment. Thus, since there are two dimensions, four higher-resolution tiles are required to produce each next lower resolution tile.

Thereafter, the tile manager **192** enters a FOR-loop at a loop state **1152**. For each of the four higher-resolution tiles, the tile manager **192** tests whether the tile is loaded in the image data cache, at a decision state **1154**. If the tile is not loaded, then the tile manager **192** moves to a state **1156** wherein a recursive call is made to the "load subimage tile" function to create the corresponding higher-resolution tile from yet higher-resolution tiles. This case occurs if a the tile is a few layers down in the image stack and the tiles in all but the full resolution subimage had been invalidated. Therefore, the function **436** invokes itself to work all the way back up to the top level, recreate the higher-resolution tiles and then work back down to the tile of interest. Only higher-resolution tiles that map to the particular lower-resolution tile need be loaded

Assuming that all the higher-resolution tiles have been loaded, the FOR-loop terminates and the tile manager **192** proceeds to test whether all of the higher-resolution tiles are blank. If all four of the high resolution tiles mapped to this low resolution are blank, the tile manager **192** transitions to a state **1160** to mark the low resolution tile as blank. The tile manager **192** does not create any image data for the blank, lower-resolution tile. The tile manager **192** and terminates the function **436** at the end state **1144**.

If, however, one or more of the higher-resolution tiles is not blank, the tile manager **192** moves to a state **1162** to make a determination as to whether it is faster to create the lower-resolution tile by scaling the com-

pressed version of the higher-resolution tiles or the expanded version of the higher-resolution tiles. An algorithm is used at the state **1162** to decide which is faster and depends on the machine that the program is running on, and other considerations. If it is faster to scale the compressed data the tile manager **192** moves to the function **442** to create the compressed, lower-resolution tile directly from the compressed higher-resolution tiles.

Now, if it is determined that it is faster to scale the expanded version of the data, the tile manager **192** moves from the state **1162** to a state **1166** to allocate memory for the uncompressed version of the lower-resolution tile. From the state **1166**, the tile manager **192** moves to the beginning of a FOR-loop at a loop state **1168** wherein for each of the higher-resolution tiles the tile manager **192** scales the expanded version of the higher-resolution tile directly into the proper position in the lower-resolution tile using the function **444**. When the tile manager **192** has scaled each of the four high resolution tiles, the tile manager **192** has completed the creation of the expanded version of the low resolution tile.

The tile manager **192** then proceeds, from either of the states **1168** or **442** to a decision state **256** wherein the tile manager **192** determines if an error was incurred in that process. If there was an error, the tile manager **192** moves to a state **1172** to report the error. From either of the states **1170** (if no error) or **1172**, the tile manager terminates the function **436** at the end state **1144**.

FIG. **32** contains the flow diagram for the "allocate space for uncompressed version of tile" function **438** referred to in FIG. **28**. The tile manager **192** enters the function **438** at a start state **1180** and moves to a decision state **1182** to test whether the "soft" uncompressed cache usage limit is exceeded. The soft uncompressed cache limit is a number that is cast into the tile manager **192** during initialization and it basically sets a guideline for how much of the image data cache is to be devoted to uncompressed image data. If the cache manager gets a request for uncompressed cache space and finds that this soft limit has been exceeded, it attempts to reduce the amount of expanded image data that is held in cache either by compressing expanded tiles or by discarding expanded tiles that have valid compressed versions or some other way to recreate them.

If the tile manager **192** finds that the soft limit is exceeded, the tile manager **192** moves to a state **1184** to first check whether the "cache collection delay" flag is set. This flag is set after an unsuccessful attempt to reduce cache memory usage and prevents repeated unsuccessful calls to collect free cache at a state **1186**.

Thus, the tile manager **192** will not try to reduce the expanded memory usage until the flag is cleared in the "unlock expanded tile handle" function **432** (FIG. **30**).

If the cache collection delay flag is not set, the tile manager moves to a state **1186** to collect free cache memory by freeing uncompressed tiles. After that, the tile manager **192** moves to a decision state **1188** to test whether the soft uncompressed cache usage limit is still exceeded after an attempt to reduce the memory usage. If the usage is still exceeded, the tile manager **192** prints a warning message on the video display **154** (FIG. **6**) at a state **1190** and then sets the cache collection delay flag at a state **1192**.

Returning in the discussion to the state **1182**, if the soft limit was not exceeded, or if it was not exceeded at the state **1188**, the tile manager **192** moves to a decision state **1194** to determine whether there is memory available in the uncompressed tile free list. If there is not memory available in the uncompressed tile free list, then the tile manager **192** moves to a decision state **1196** to determine whether there is memory available in the cache reserve list. If there is no memory available there, the tile manager **192** moves to a state **329** wherein the tile manager **192** again tries to collect free cache space by unlocking or freeing both uncompressed and compressed tiles. At this point, the tile manager **192** must free space in order to allocate space for this uncompressed tile. The tile manager **192** moves to a state **1200** to determine whether memory is now available in the cache reserve list. In the state **1198**, when the cache memory space is freed, it is placed into the cache reserve list. If memory is not available, then the tile manager **192** moves to a state **1202** and prints a "cache overflow" error message and terminates the function **438** with an error condition at the end state **1204**.

Now, taking an alternate path from the states **1194**, **1196** and **1200**, if the tile manager **192** can successfully get space for the uncompressed tile data, then the tile manager **192** moves to a state **1206** where the tile manager **192** finds the free block with the highest memory address. If there is a choice between two or more free memory blocks, the tile manager **192** chooses the one with the highest address to try to keep all of the expanded image data at the high address end of the cache file. Once the tile manager **192** finds the highest address block, it moves to a state **1208** to unlink the free block from the free memory link list.

There are actually two possibilities for the free memory link list when the tile manager **192** is looking for expanded memory. One is the uncompressed tile free list and the other is the cache reserve list. In either case, the tile manager **192** unlinks the block of memory that the tile manager **192** is interested in from the free list and relinks the remaining memory blocks of the affected free list.

The tile manager **192** then transitions to a state **1210** to initialize the newly allocated block to all background color. Then the tile manager **192** moves to a state **1212** to move the description of the memory block (a pointer to the tile header) to the front of the most recently used tile list. Moving to a state **1214**, the tile manager **192** updates the soft uncompressed cache memory usage counter that was checked at the state **1182**. The tile manager **192** continues to a state **1216** to store the memory address in the tile header. The memory block that the tile manager **192** has just allocated is a pointer that is stored in the tile header data structure. That is how the memory block is associated with the tile. Then the tile manager **192** terminates normally from the function **438** at the end state **1204**.

FIG. **33** illustrates the process by which the present invention expands the compressed version of a tile to create an uncompressed version. Specifically, as shown in FIG. **33**, the tile manager **192** starts at a start state **1220** and moves to a test function at state **1222**, where the tile manager **192** determines whether the compressed version of the tile, or the compressed tile data, is in cache memory. If it is not, then the tile manager **192** moves to state **1224**, wherein the system loads the necessary data from the disk. If there is an error detected at state **1224**, the tile manager **192** moves to state **1228** to terminate the process.

From state **1226**, if compressed data was successfully loaded from the disk or from state **1222** if it was in cache

to begin with, the tile manager **192** moves to state **1230**, wherein the tile manager **192** locks the compressed tile image data. This step simply increments the lock count on the compressed memory state. From state **1230**, the system moves to state **1232**, wherein the tile manager **192** allocates and locks the uncompressed tile memory block. The system then moves to state **1234** to determine whether an error occurred at state **1232**. If so, the tile manager **192** moves to state **1236** and unlocks the compressed tile data. From state **1236**, the system moves to state **1238** to report the error. The system then terminates at end state **1228**.

On the other hand, if no error existed as determined at state **1234**, the system moves to state **1240**, wherein the tile manager **192** uncompresses the compressed data. Next, the tile manager **192** moves to state **1242** to determine whether an error occurred at state **1240**. If an error occurred at state **1240**, the tile manager **192** moves to state **1236** and functions as described previously. Otherwise, the tile manager **192** moves to stat **1244** to unlock the compressed and uncompressed data, and then terminates at end state **1228**.

FIG. **34** illustrates a process for creating compressed low resolution tiles from compressed higher resolution tiles. The tile manager **192** starts at start state **1250** and proceeds to state **1252**, wherein the system enters a loop which is followed by the system for each of the four high resolution tiles required to produce a single low resolution tile. More specifically, at state **1252** the tile manager **192** locks the compressed version of the high resolution tile. The system then proceeds to state **1256**, wherein the tile manager **192** determines whether an error occurred at state **1254**. In the event that an error occurred, the tile manager proceeds to end state **1258** and terminates. If no error occurred, the tile manager **192** returns to state **1252** and continues the loop described above for each of the four high resolution tiles.

After processing all four high resolution tiles as described, the system proceeds to state **1260** where the tile manager **192** scales the compressed data to half resolution. The process performed at state **1260** results in a compressed version of the low resolution tile. Then the tile manager **192** moves to a loop represented by states **1262, 1264**, wherein for each of the high resolution tiles the tile manage **192** unlocks the compressed version of the tile.

Next, the tile manager **192** moves to state **1266** where the tile manager **192** allocates and locks memory for the compressed version of the low resolution tile. At state **1266**, the tile manager **192** actually puts the compressed version of the low resolution tile in a general, common buffer that is large enough to hold the maximum possible size of the compressed results. The actual valid data is usually much less than that than the maximum possible size, so the tile manager **192** only saves the valid amount of data.

From state **1266**, the system moves to state **1268** to determine whether an error occurred at state **1266**. If an error occurred, the system moves to end state **1258** and terminates. Otherwise, the system moves to state **1270** where the tile manager **192** copies the compressed data out of the temporary compressed data buffer into the newly allocated space in the cache. Then the tile manager **192** moves to state **1272** where the tile manager **192** unlocks the compressed version of the low resolution tile that now contains valid data. The system then terminates normally at state **1258**.

Now referring to FIG. **35**, a process is shown whereby the system resamples uncompressed high resolution tiles to an uncompressed low resolution tile. The tile manager **192** starts at start state **1280** and moves to state **1282**, wherein the tile manager **192** locks the uncompressed version of a single high resolution tile. This function scales a single high resolution tile to update one quarter of a tile in the half-resolution subimage. That quarter tile is rescaled to update one-sixteenth of a tile in the quarter-resolution subimage. This continues to the lowest resolution subimage. Next, the tile manager **192** proceeds to state **1284** to determine whether an error occurred in locking the uncompressed version of the high resolution tile. If there Was an error, then the tile manager **192** proceeds to state **1286** and terminates with an error condition. Otherwise, the tile manager **192** moves to state **1288** where the tile manager **192** determines how many levels of the subimage are to be updated. This function can be used to update a subset of subimages or the entire image stack in the case where a single tile is modified in the full resolution subimage. It will propagate that change all the way down to the lowest-resolution subimage in the image stack.

Next, the tile manager **192** proceeds to state **1290** where the tile manage **192** determines the tile index that is to be updated. In accordance with the present invention, when a change is propagated from the higher resolution down to the low resolution of tiles, the system calculates which tile corresponds to the affected area. Then the tile manager **192** moves to state **1290** where the tile manager **192** determines whether the low resolution tile that the tile manager **192** is about to update is marked as loaded or not. This step is intended for the situation in which not all of the low resolution substates are populated during the loading of a raster image.

If the system determines that one or more low resolution tiles are not loaded, the system proceeds to state **1294**, wherein the tile manager **192** invalidates all of the low resolution tiles that would otherwise be affected by the change. The system then exits normally at end state **1286**. If the low resolution tile is about to be modified is loaded, as determined at state **1292**, the tile manager **192** moves to state **1296**, wherein the system locks the uncompressed version of the low resolution tile. The tile manager **192** then moves to state **1298** to determine whether an error occurred at state **1296** and, if so, the system moves to end state **1286** to terminate. Otherwise, the system moves to state **1300**. wherein the tile manager **192** scales the raster data from the high resolution tile down to the low resolution tile. Then the tile manager **192** moves to state **1302** where the tile manager **192** unlocks the high resolution tile.

Next, the system moves to state **1304**, wherein the tile manager **192** recursively modifies the loop variables such that the low resolution tiles that the tile manager **192** just finished updating become the high resolution tiles for the next succeeding iteration. Once all the subimages have been updated as described, the system exits at end state **1286**.

Now referring to FIGS. **36A** and **36B**, a process to collect free cache is shown. This process can be called from several other processes. The tile manager **192** begins at start state **1310** in FIG. **36A** and moves to state **1312** to determine whether a cache collection operation is in process. If so, the system exits at end state **1314**. This prevents recursive calls to collect free cache which might otherwise occur. If the system at state

1312 determines that no collection is in progress, then the tile manager 192 moves to state 1316 where the tile manager 192 sets a flag indicating that a collection is in progress.

From state 1316, the system moves to state 1320, where the tile manager 192 estimates the number of memory blocks to free in this operation. The reason for freeing a number of blocks instead of just one block is to reduce the computational overhead associated with the cache collection operations. The tile manager 192 typically estimates the amount of memory required to equal the number of tiles in a single row of the full resolution subimage of the document associated with the most recently used tile.

Once this estimate has been made, the system proceeds to state 1322 wherein the tile manager 192 considers the options that the tile manager 192 passed into this function. There are three options. One, as indicated at state 1324, is to reduce the uncompressed cache usage only while not affecting the compressed data that is currently held in cache. The second option, indicated at state 1328, is to reduce the compressed cache memory usage only. The third option, indicated at state 1326, is to reduce the total cache memory usage including both compressed and uncompressed data.

From state 1324 or state 1326, the tile manager 192 moves to state 1330, where the tile manager 192 stores all of the free states currently in the uncompressed free list into the cache reserve list. As the tile manager 192 performs the process in state 1330, the tile manager 192 attempts to consolidate the memory blocks. That is, if there are two free blocks that are adjacent to one another, the system automatically turns them into a single, larger contiguous block. From state 1328, on the other hand, the system moves to state 1358, shown in FIG. 36B and discussed below.

From state 1330, the tile manager 192 moves to state 1332, wherein the tile manager 192 determines whether the tile manager 192 has created a memory block large enough to satisfy the initial request. If so, the tile manager 192 terminates normally at end state 1314. Otherwise, the tile manager 192 moves to state 1334 where the tile manager 192 frees any unlocked, uncompressed tiles which are blank. The tile manager 192 then moves to state 1336 where the tile manager 192 determines whether the tile manager 192 has free sufficient memory. If so, the tile manager 192 exits at end state 1314. Otherwise, the tile manager 192 moves to state 1338 where the tile manager 192 frees unlocked, unpreserved uncompressed tiles that have valid compressed versions in cache or are on a disk, or that have valid, uncompressed versions on the disk beginning with the least recently used tile. After having freed that particular class of tiles, if the tile manager 192 determines, at state 1340, that the memory request has been satisfied, the tile manager 192 moves to state 1314 and terminates. Otherwise, the tile manager 192 moves to state 1342, shown in FIG. 36B.

Now referring to FIG. 36B, the tile manager 192 begins at state 1342, wherein the tile manager 192 compresses the free unlocked, unpreserved uncompressed tiles that don't have a valid compressed version or other source from which the tile can be recreated. To do this the tile manager 192 processes expanded tile data through a compression algorithm. The tile manager 192 then creates a compressed version of that tile so that the uncompressed version of the tile can be discarded.

Next, the tile manager 192 moves to state 1344, wherein the system determines whether the request made at state 1342 has been satisfied. If so, the system terminates at end state 1346. Otherwise, the system moves to state 1348, wherein the tile manager 192 frees unlocked, but preserved uncompressed tiles that have valid compressed or uncompressed copies. The tile manager 192 preferentially frees the oldest such tiles.

From the state 1348, the tile manager 192 proceeds to a decision state 1350 to test whether the request made at the state 1348 was satisfied. If so, the function 446 is terminated at the end state 1346. Otherwise, the tile manager 192 moves to a state 1352 to compress and then free unlocked, but preserved, uncompressed tiles that do not have valid compressed versions.

Next, the tile manager 192 moves to state 1354, wherein the system determines whether the request made at state 1352 has been satisfied. If so, the system terminates at end state 1346. Otherwise, the system moves to state 1356, wherein the tile manager 192 determines whether to free data memory blocks. If not, the system terminates at state 1346. Otherwise, the system moves to state 1358, to free unlocked preserved, uncompressed tiles that don't have valid compressed versions already.

The system next moves to state 1360 to determine whether the request has been satisfied. If so, the system terminates at state 1346. Otherwise, the system moves to state 1362 to print an error message, and then terminate at state 1346.

Now referring to FIG. 37, the tile manager 192 starts at state 1380 and moves to state 1382 where the tile manager 192 determines whether the uncompressed version is in fact still locked—that is if the lock count for uncompressed version of that tile is non-zero. If the tile is still locked then the tile manager 192 moves to state 1384 and prints a warning message. Then the tile manager 192 terminates at end state 1386.

If, at state 1382, the system determined that the uncompressed version is not locked, then the tile manager 192 moves to state 1388 where the tile manager 192 determines whether the uncompressed data has already been freed. If it has then the tile manager 192 terminates at end state 1386. Otherwise, the tile manager 192 moves to state 1390 where the tile manager 192 unlinks the uncompressed memory state from the most recently used list.

From state 1390, the tile manager 192 moves to state 1392 where the tile manager 192 updates and decrements the total uncompressed memory usage counter by the appropriate amount. The tile manager 192 then moves to state 1394 where the tile manager 192 moves the memory block to the uncompressed memory free list. In accordance with the present invention, the tile manager 192 keeps the list sorted by decreasing address. Consequently, when the tile manager 192 allocates expanded memory blocks, the tile manager 192 tends to choose the preferred blocks that have higher addresses because they are at the front of the free list.

Next, the tile manager 192 moves to state 1396, wherein the tile manager 192 sets a pointer in the tile header to null and the tile manager 192 sets the uncompressed tile status flags. This ensures that the tile header reflects the fact that it no longer has an uncompressed data associated with it. Then the tile manager 192 terminates at end state 1386.

Now referring to FIG. 38, a process by which the system compresses a tile is shown. The system begins at

start state 1400, and moves to state 1402, wherein the tile manager 192 determines whether the uncompressed tile data is in cache memory. If it is not, the tile manager 192 moves to state 1404 and loads the uncompressed data into cache memory from the disk. The system then moves to state 1406, to determine whether an error occurred at state 1404. If so, the system terminates at end state 1408. Otherwise, the system proceeds to state 1410.

At state 1410, the tile manager 192 locks the uncompressed tile data, and then moves to state 1412, to determine whether an error occurred at state 1410. If an error occurred, the system terminates at end state 1408. Otherwise, the system moves to state 1414, wherein the tile manager 192 compresses the image data into a common buffer. For binary images of text and line drawings, the tile manager 192 uses a CCITT group 4 encoding.

From state 1414, the tile manager 192 moves to state 1416 to determine whether an error occurred at state 1414. If an error indeed occurred, the system moves to state 1418 to unlock the uncompressed tiles, and then exits at end state 1408. Otherwise, the system proceeds to state 1420, wherein the tile manager 192 allocates and locks cache memory space for the compressed tile data.

From state 1420, the system proceeds to state 1422 to determine whether an error occurred at state 1420. If an error occurred, the system moves to state 1418 and proceeds as described above. Otherwise, the system moves to state 1424, wherein the tile manager 192 copies the compressed data from the common buffer into the newly allocated cache memory state. The system moves from state 1424 to state 1426, wherein the tile manager 192 unlocks the compressed and uncompressed tile data and then terminates at end state 1408.

While the above detailed description has shown, described and pointed out the fundamental novel features of the invention as applied to various embodiments, it will be understood that various omissions and substitutions and changes in the form and details of the device illustrated may be made by those skilled in the art, without departing from the spirit of the invention.

What is claimed is:

1. An image memory management system, comprising:
- a computer having a processor and an image memory, the image memory comprising a main memory and a secondary memory;
- an image stack, located in the image memory, comprising a plurality of similar digital images, each digital image having a plurality of pixels grouped into at least one tile, and each digital image having a resolution different from the other digital images;
- means for accessing a selected one of the tiles in the image stack;
- first means for transferring a selected one of the tiles from the secondary memory to the main memory when the tile is accessed by the accessing means and the tile is absent from the main memory; and
- second means for transferring a selected one of the tiles from the main memory to the secondary memory when the main memory is full.

2. The system defined in claim 1, additionally comprising means for modifying a selected one of the tiles.

3. The system defined in claim 2, wherein the second transferring means only transfer tiles that have been modified by the modifying means.

4. The system defined in claim 1, wherein the main memory is semiconductor memory.

5. The system defined in claim 1, wherein the secondary memory is a magnetic disk.

6. The system defined in claim 1, wherein each tile is square.

7. The system defined in claim 1, wherein a lowest resolution digital image comprises one tile.

8. The system defined in claim 1, wherein a preselected digital image in the image stack is resampled to obtain another digital image in the image stack.

9. The system defined in claim 1, wherein at least one of the digital images is compressed.

10. The system defined in claim 1, wherein the accessing means is responsive to an image access operation selected by a user.

11. The system defined in claim 10, wherein the image access operation is zooming or panning the image.

12. The system defined in claim 10, wherein the image access operation is reversible.

13. A method of managing images in a computer having a processor and an image memory comprising a slower access memory and a faster access memory, comprising the steps of:
- creating a digital image;
- resampling the digital image so as to form an image stack comprising the digital image and one or more lower resolution digital images;
- dividing each image into equal sized, rectangular tiles; and
- evaluating a location in the image memory of tiles in each digital image of the image stack in a given region of interest.

14. The method defined in claim 13, additionally comprising updating modified regions of all images when an edit operation is completed.

15. The method defined in claim 13, wherein the evaluating step includes the following order of decreasing availability:
- exists in the faster access memory in uncompressed form;
- exists in the slower access memory in uncompressed form;
- exists in the faster access memory in compressed form;
- exists in the slower access memory in compressed form; and
- must be constructed from higher resolution tiles.

16. The method defined in claim 13, wherein the evaluating step includes the following order of decreasing availability:
- exists in the faster access memory in uncompressed form;
- exists in the slower access memory in uncompressed form;
- exists in the slower access memory in compressed form; and
- must be constructed from higher resolution tiles.

17. The method defined in claim 13, wherein the evaluating step includes selecting the digital image with the lowest resolution higher than a requested resolution at a given view scale.

* * * * *

United States Patent [19]

Delorme

[11] **Patent Number: 4,972,319**

[45] **Date of Patent: Nov. 20, 1990**

[54] **ELECTRONIC GLOBAL MAP GENERATING SYSTEM**

[76] Inventor: **David M. Delorme**, 356 Range Rd., Cumberland, Me. 04021

[21] Appl. No.: **101,315**

[22] Filed: **Sep. 25, 1987**

[51] **Int. Cl.**[5] **G09B 29/00**

[52] **U.S. Cl.** **364/419**; 434/150; 340/990

[58] **Field of Search** 364/419, 449; 434/150, 434/130; 340/990

[56] **References Cited**

U.S. PATENT DOCUMENTS

400,642	4/1889	Beaumont	283/34
751,226	10/1899	Van Der Grinten	283/34
752,957	2/1904	Colas	283/34
1,050,596	1/1913	Bacon	283/34
1,610,413	12/1924	Balch	283/34
2,094,543	9/1937	Lackey et al.	353/11
2,354,785	8/1944	Von Rohl	434/150
2,431,847	12/1947	Dusen	353/11
2,650,517	9/1953	Falk	355/77
3,248,806	5/1966	Schrader	434/150
3,724,079	4/1973	Jasperson et al.	33/15 B
4,315,747	2/1982	McBryde	434/150
4,673,197	6/1987	Stipelman et al.	434/150
4,689,747	8/1987	Krouse et al.	364/449
4,737,927	4/1988	Hanabusa et al.	340/990

OTHER PUBLICATIONS

"Equal-Area Projections for World Statistical Maps", McBryde and Thomas, U.S. Dept. of Commerce, Coast and Geodetic Survey, Spec. Pub. 245, 1949.

"The Quadtree and Related Hierarchical Data Structures", Hanan Samet, Computer Surveys, vol. 16, No. 2, Jun. 1984.

Primary Examiner—Jerry Smith
Assistant Examiner—Kim T. Bui
Attorney, Agent, or Firm—Sughrue, Mion, Zinn, Macpeak & Seas

[57] **ABSTRACT**

A global mapping system which organizes mapping data into a hierarchy of successive magnitudes or levels for presentation of the mapping data with variable resolution, starting from a first or highest magnitude with lowest resolution and progressing to a last or lowest magnitude with highest resolution. The idea of this hierarchical structure can be likened to a pyramid with fewer stones or "tiles" at the top, and where each successive descending horizontal level or magnitude contains four times as many "tiles" as the level or magnitude directly above it. The top or first level of the pyramid contains 4 tiles, the second levle contains 16 tiles, the third contains 64 tiles and so on, such that the base of a 16 magnitude or level pyramid would contain 4 to the 16th power or 4,294,967,296 tiles. This total includes "hyperspace" which is later clipped or ignored. Digital data corresponding to each of the separate data base tiles is stored in the database under a unique filename.

33 Claims, 9 Drawing Sheets

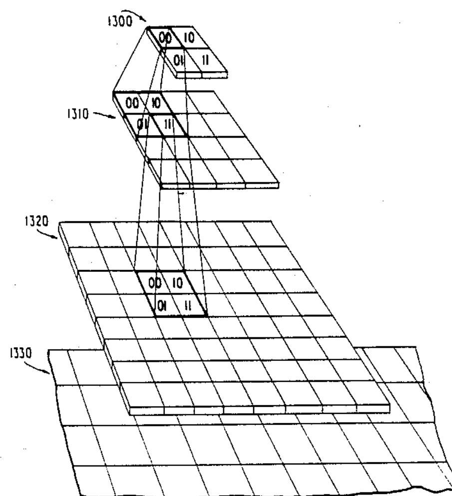


FIG. 1

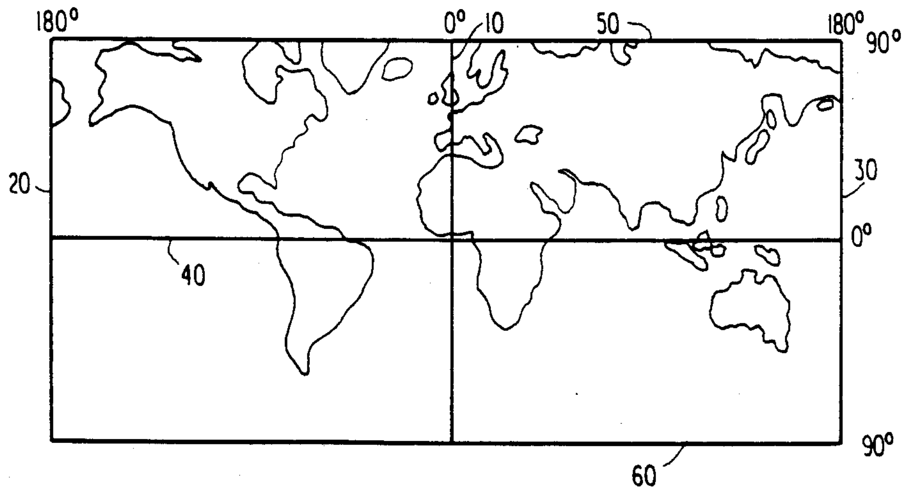


FIG. 2

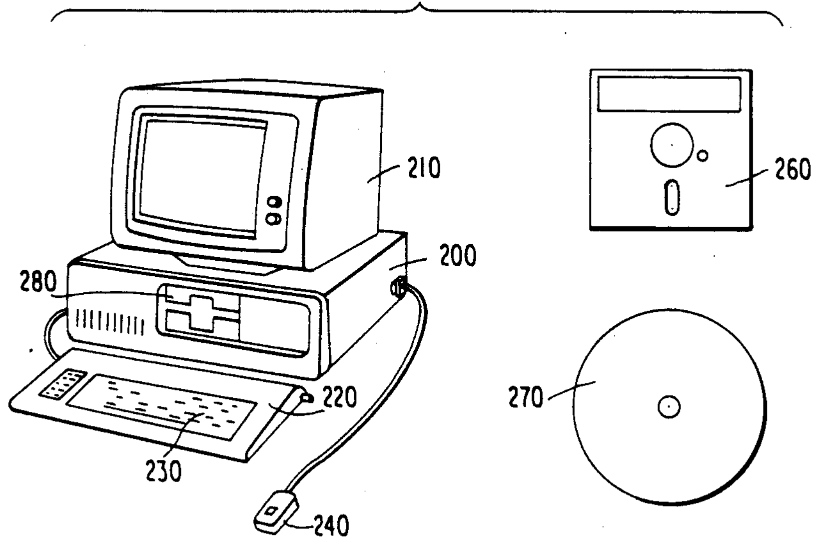


FIG. 3A

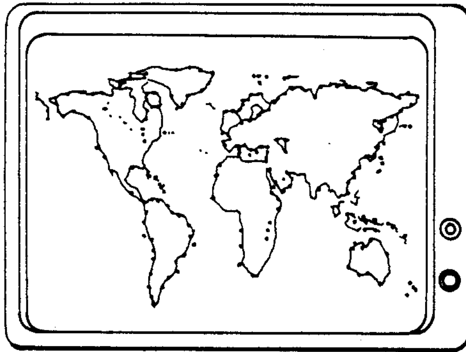


FIG. 3B

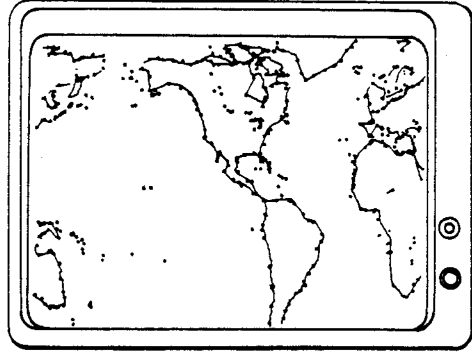


FIG. 3C

FIG. 3D

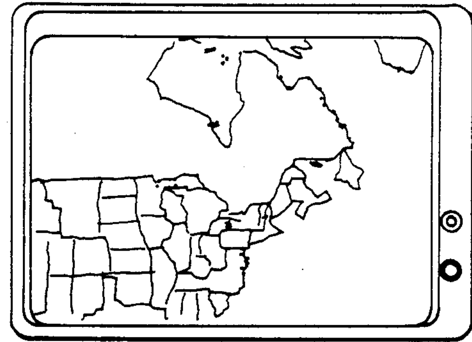


FIG. 3E

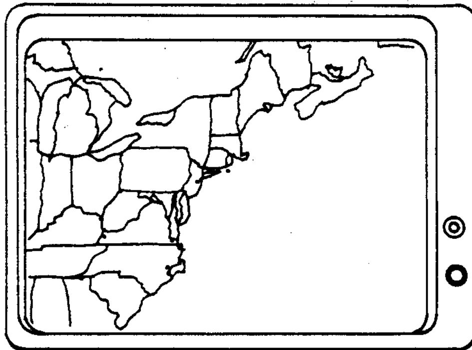


FIG. 3F

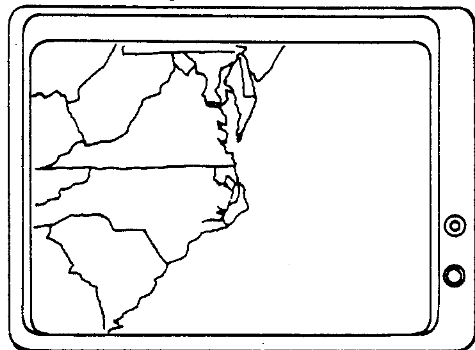


FIG. 4

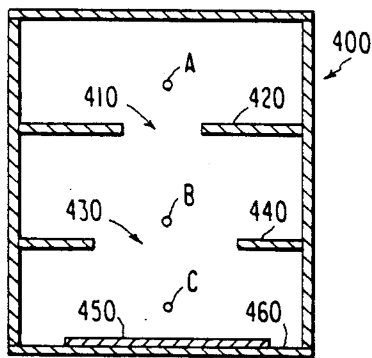


FIG. 5A

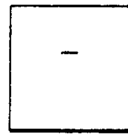
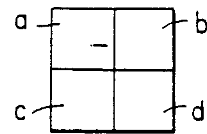


FIG. 5B



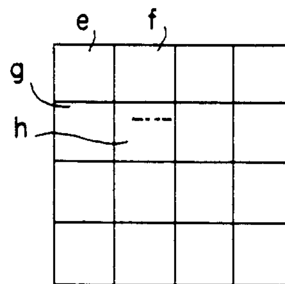
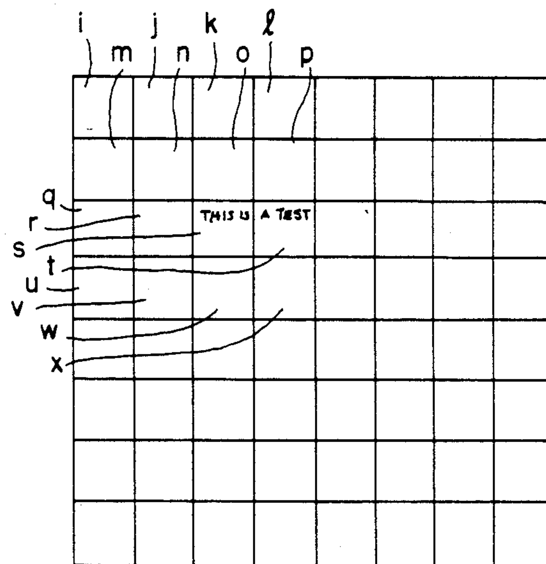


FIG. 6

FIG. 7



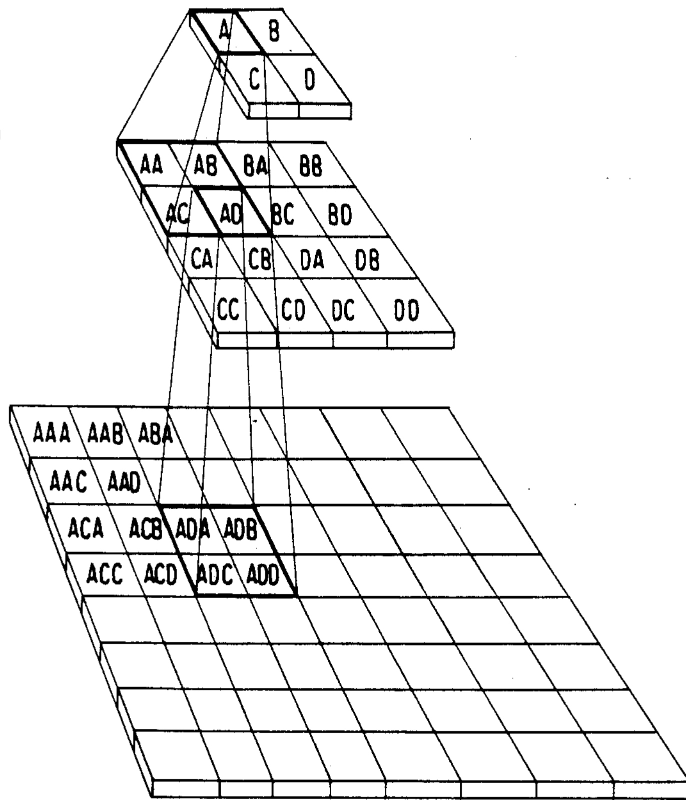


FIG. 8

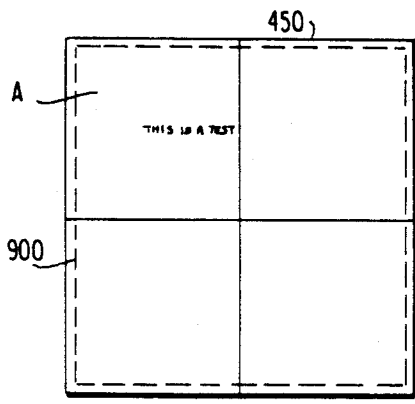


FIG. 9A

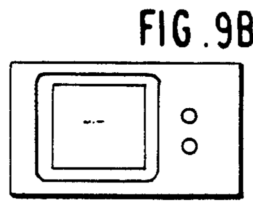


FIG. 9B

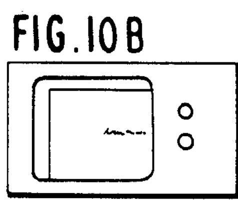


FIG. 10B

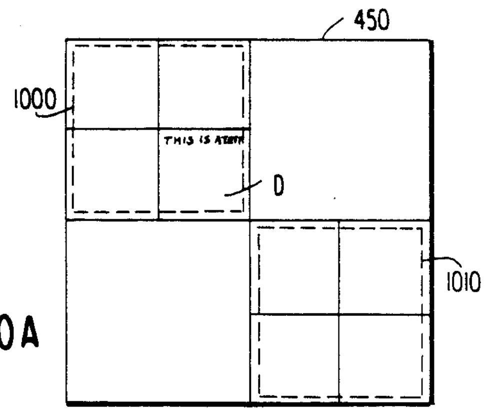


FIG. 10A

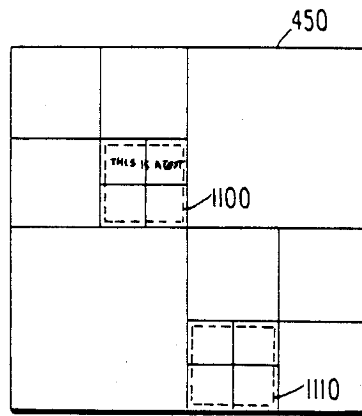


FIG. 11A

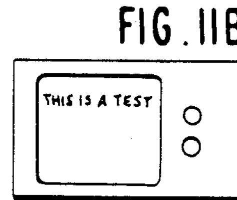


FIG. 11B

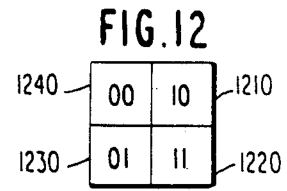


FIG. 12

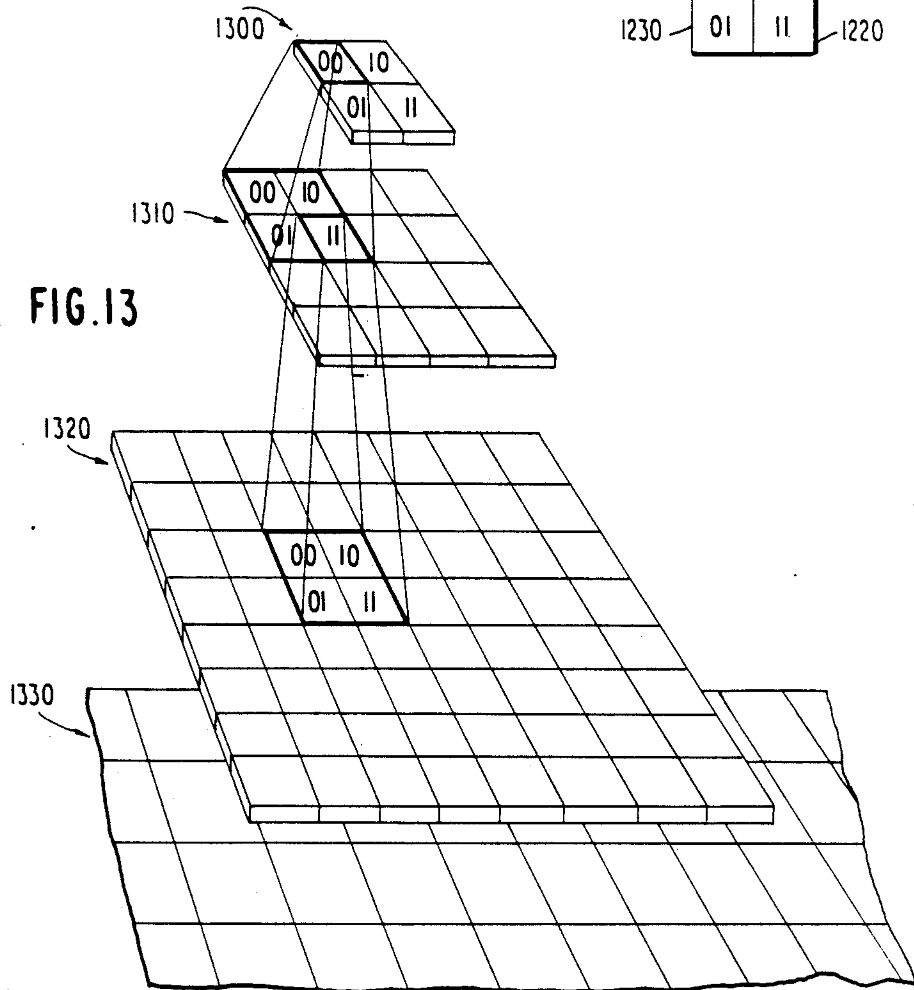


FIG. 13

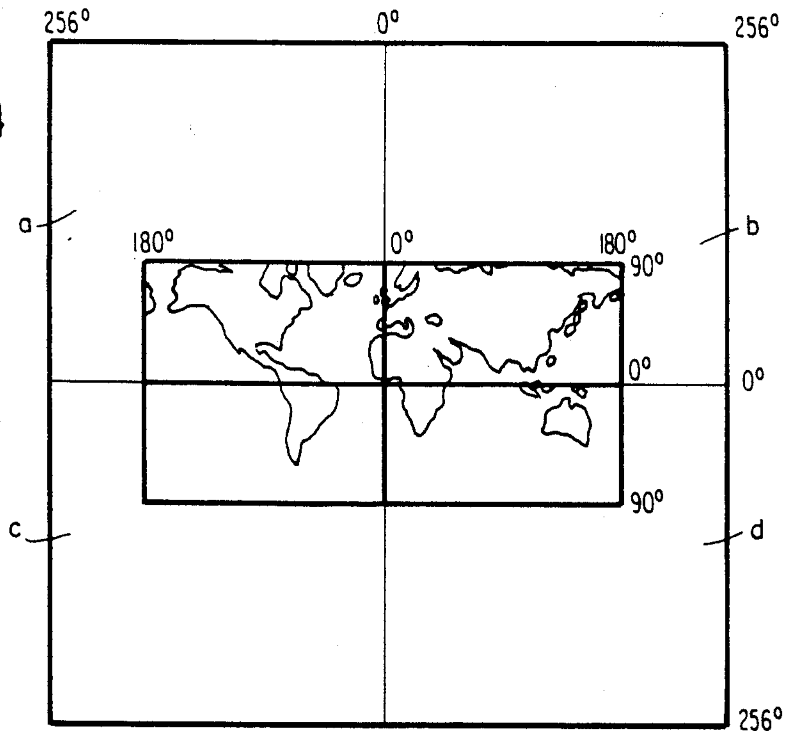


FIG.14
256°
0°
256°
a
b
180°
0°
180°
90°
0°
0°
90°
c
d
256°

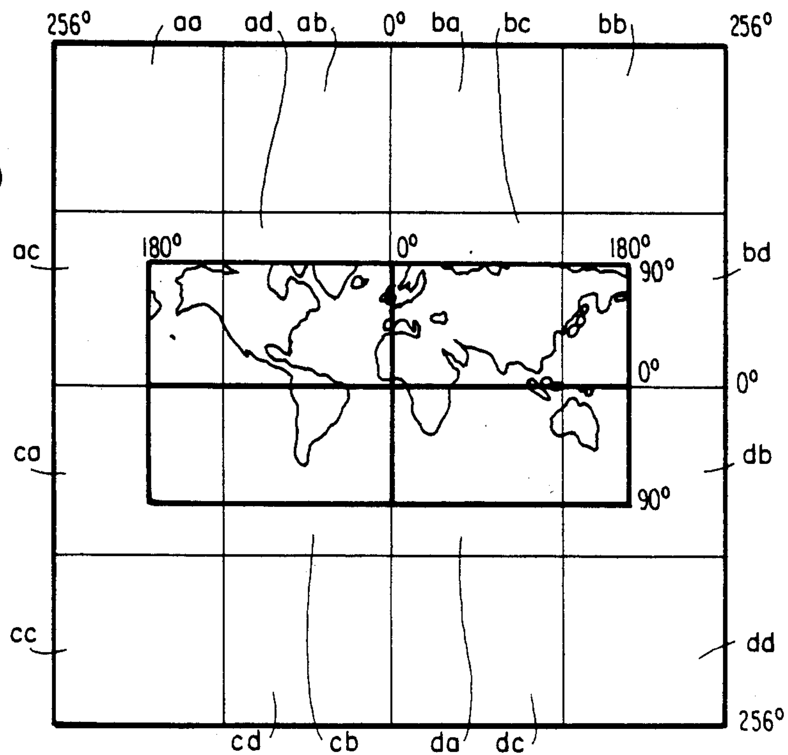


FIG.15
256°
aa
ad
ab
0°
ba
bc
bb
256°
ac
180°
0°
180°
90°
bd
0°
0°
ca
db
90°
cc
dd
256°
cd
cb
da
dc

FIG. 16

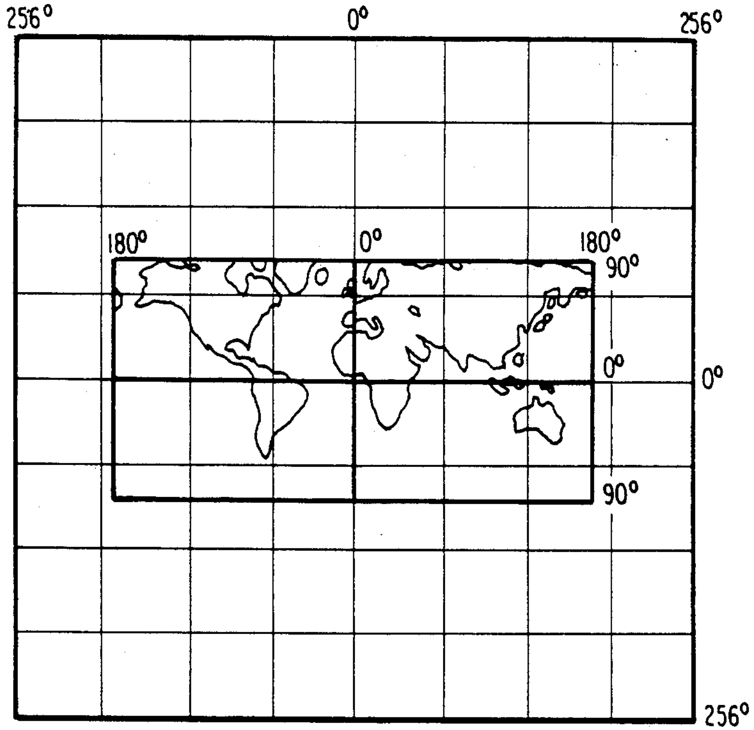
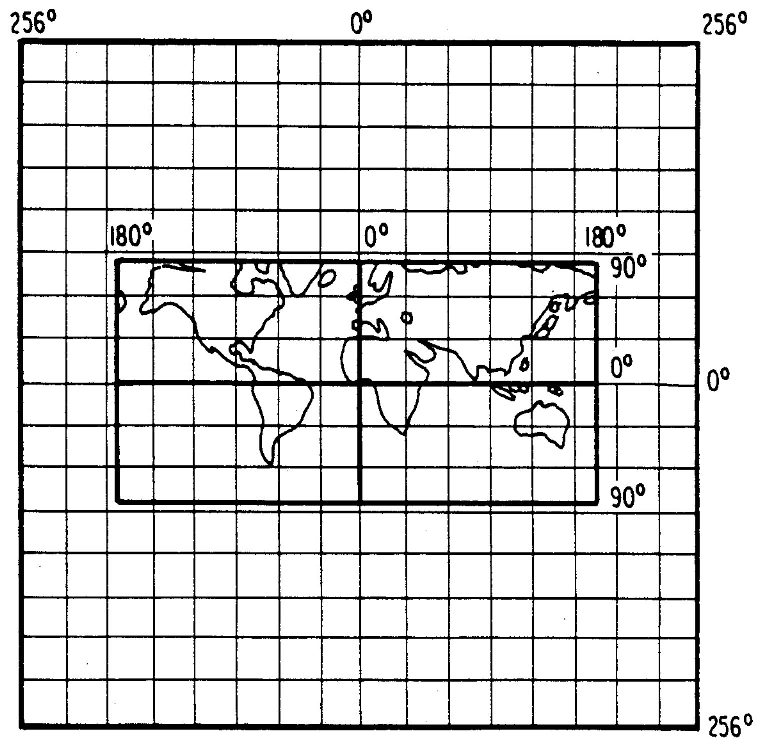


FIG. 17



 Unified Patents Exhibit 1005 App'x A-N

FIG. 18

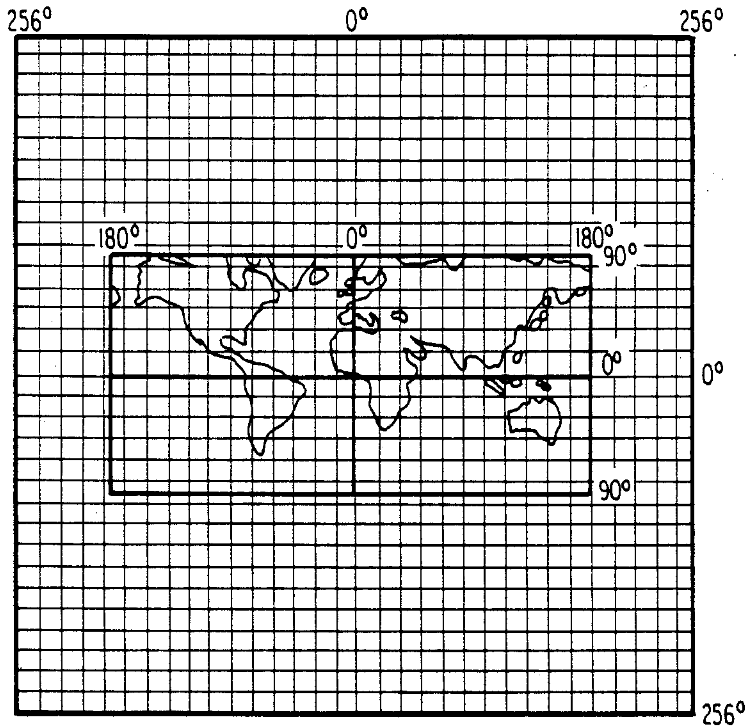


FIG. 19

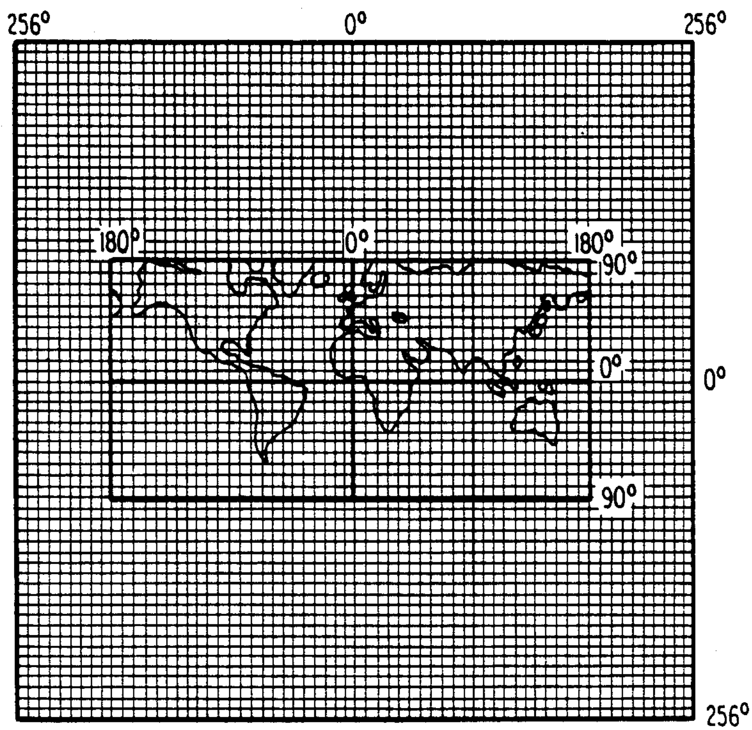
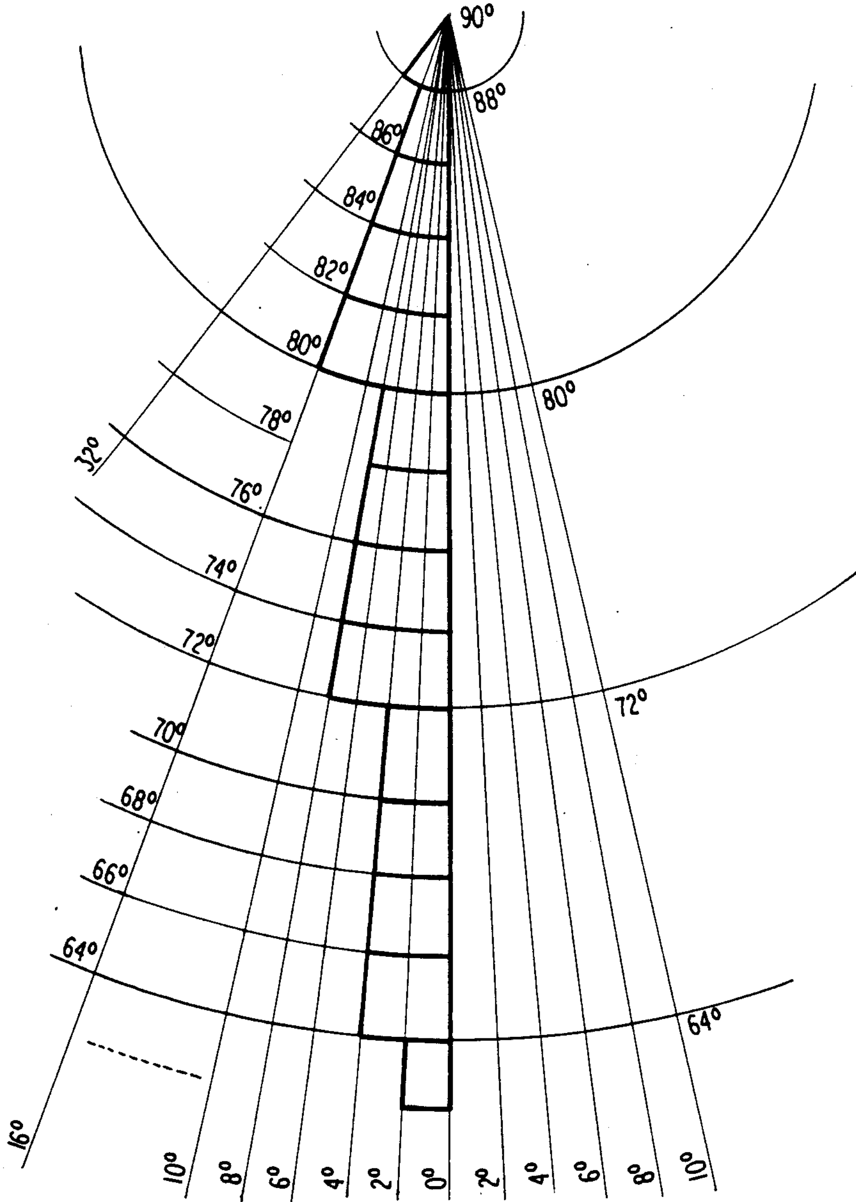


FIG. 20

ILLUSTRATION OF POLAR COMPRESSION AT THE 8th MAGNITUDE



ELECTRONIC GLOBAL MAP GENERATING SYSTEM

BACKGROUND OF THE INVENTION

1. Technical Field

This invention relates to a new variable resolution global map generating system for structuring digital mapping data in a new data base structure. managing and controlling the digital mapping data according to new mapping data access strategies, and displaying the mapping data in a new map projection of the earth.

2. Background Art

Numerous approaches have been forwarded to provide improved geographical maps, for example:

U.S. Pat. No. 4,315,747, issued to McBryde on Feb. 16, 1982, describes a new map "projection" and intersecting array of coordinate lines known as the "graticule", which is a composite of two previously known forms of projection. In particular, the equatorial portions of the world are represented by a fusiform equal area projection in which the meridian curves, if extended, would meet at points at the respective poles, referred to as "pointed poles". In contrast, the polar regions of the world map are represented by a flat polar equal area projection in which the poles are depicted as straight horizontal lines with the meridians intersecting along its length. Thus, in a flat polar projection the meridian curves converge toward the poles but do not meet at a point and, instead, intersect a horizontal linear pole. The two component portions of the flat world map are joined where the parallels are of equal length. The composite is said to be "homolinear" because all of the meridian curves are similar curves, for example, sine, cosine or tangent curves, which merge where the two forms of projection are joined where the respective parallels are equal. The flat polar projections in the polar portions of the map provide a compromise with the Mercator cylinder projections, thereby greatly reducing distortion.

U.S. Pat. No. 1,050,596, issued to Bacon on Jan. 14, 1913, describes another composite projection for world maps and charts which uses a Mercator or cylindrical projection for the central latitudes of the earth and a convergent projection at the respective poles. In the central latitudes, the grids of the Mercator projection net or graticule are rectangular. In the polar regions, the converging meridians may be either straight or curved.

U.S. Pat No 1,620,413, issued to Balch on Dec. 14, 1926, discusses gnomic projections from a conformal sphere to a tangent plane and Mercator or cylindrical projections from the conformal sphere to a tangent cylinder. Balch is concerned with taking into account the non-spherical shape of the earth, and therefore, devises the so-called "conformal sphere" which represents the coordinates from the earth whose shape is actually that of a spheroid or ellipsoid of revolution, without material distortion.

U.S. Pat. No. 752,957, issued to Colas on Feb. 23, 1904, describes a map projection in which a map of the entire world is plotted or transcribed on an oval constructed from two adjacent side by side circles with arcs joining the two circles. The meridians are smooth curves equally spaced at the equator, while the latitude lines are non-parallel curves.

U.S. Pat. No. 400,642 issued to Beaumont on Apr. 2, 1889, describes a map of the earth on two intersecting spheres, on which the coordinate lines of latitude and longitude are all arcs of circles.

U.S. Pat. No. 751,226, issued to Grinten on Feb. 2, 1904, represents the whole world upon the plane surface of a single circle with twice the diameter of the corresponding globe, the circle being delineated by a graticule of coordinates of latitude and longitude which are also arcs of circles.

U.S. Pat. No. 3,248,806, issued to Schrader on May 3, 1966, discloses a subdivision of the earth into a system of pivotally mounted flat maps, each map segment representing only a portion of the earth's surface in spherical projection on an equilateral spherical triangle to minimize distortion.

U.S. Pat. No. 2,094,543, issued to Lackey et al on Sept. 28, 1937, describes a projector for optically producing a variety of different map projections, including orthographic, stereographic and globular projections onto flat translucent screens and a variety of other projections on shaped screens.

U.S. Pat. No. 2,650,517, issued to Falk on Sept. 1, 1953, describes a photographic method for making geographical maps.

U.S. Pat. No. 2,354,785, issued to Rohl on Aug. 1, 1944, discloses two circular maps which are mounted side by side, and an arrangement for rotating the two maps in unison so that corresponding portions of the earth's surface are at all times in proper relationship.

U.S. Pat. No. 3,724,079, issued to Jasperson et al on Apr. 3, 1973, discloses a navigational chart display device which is adapted to display a portion of a map and enable a pilot to fix his position, to plot courses and to measure distances.

U.S. Pat. No. 2,431,847 issued to Van Dusen on Dec. 2, 1947, discloses a projection arrangement, in which a portion of the surface of a spherical or curved map may be projected in exact scale and in exact proportional relationship.

McBryde and Thomas, *Equal Area Projections for World Statistical Maps*, Special Publication No. 245, Coast & Geodetic Survey 1949.

In addition to the above further teachings as to geographical mapping can be found in the *Elements of Cartooraphy*, 4th edition which was written by Arthur Robinson, Randall Sale and Joel Morrison, and published by John Wiley & Sons (1978).

The present invention seeks to provide a low cost and efficient mapping system which allows the quick and easy manipulation of and access to an extraordinary amount of mapping information, i.e., a mapping system which allows a user to quickly and easily access a detailed map of any geographical area of the world.

Map information can be stored using at least three different approaches, i.e., paper, analog storage and digital storage, each approach having its own advantages and disadvantages as detailed below.

The paper mapping approach has been around since papyrus and will probably exist for the next thousand years.

Advantages of paper storage:

inexpensive.

once printed, no further processing is required to access the map information, so not subject to processing breakdown.

Disadvantages of paper storage:

can become bulky and unwieldy when dealing with a large geographical area, or a large amount of maps.

paper does not have the processing capabilities or "intelligence" of computers, and therefore does not support automated search or data processing capabilities.

cannot be updated cheaply and easily.

The analog mapping approach is used to provide what is commonly known as videodisc maps. The information is stored as still frames under N.T.S.C. (National Television Standards Committee) conventions. To make maps, a television camera moves across a paper map lying on a workbench. Every few inches a frame is recorded on videotape. After one row of the map is completely recorded, the camera is moved down to the next row of frames to be recorded. This process is repeated until frames representing a checkerboard pattern of the entire map are recorded. The recorded videotape could be used to view the map: however, access time to scan to different areas of the recorded map is usually excessive. As a result, a videodisc, with its quicker access time, is typically used as the medium for analog map storage. The recorded videotape is sent to a production house which "stamps" out 8 inch or 12 inch diameter, videodiscs.

Advantages of the analog storage approach:

one side of a 12 inch videodisc can hold 54.000 "frames" of a paper map. A frame is typically equal to $2\frac{1}{2}\times 3$ inches of the paper map.

access time to any frame can be fast usually under 5 seconds.

once located on the videodisc, the recorded analog map information will be used to control the raster scan of a monitor and to produce a reproduction of the map in 1/30th of a second.

through additional hardware and software, mapping symbols, text and/or patterhsn can be overlaid on top of the recorded frame.

Disadvantages of the analog storage approach:

the "frames" are photographed from paper maps, which, as mentioned above, cannot be updated cheaply or easily.

due to paper map projections, mechanical camera movements, lens distortions and analog recording electronics, the videodisc image which is reproduced is not as accurate as the original paper map.

as a result of the immediately above phenomena, latitude and longitude information which is extracted from the reproduced image cannot be fully trusted.

if a major error is made in recording any one of the 54,000 frames, it usually requires redoing and re-stamping.

since frames cannot be scrolled, most implementations employ a 50% overlap technique. This allows the viewer to jump around the database with a degree of visual continuity: however, this is at a sacrifice of storage capacity. If the frame originally covered $2\frac{1}{2}\times 3$ inches or approximately 8 square inches of the paper map, the redundant overlap information is 6 square inches, leaving only 2 square inches of new information in the centroid of each frame.

as a result of the immediately above deficiency, a 2×3 foot map containing 864 square inches would require 432 frames; thus, only 125 paper maps could be stored on one side of a 12 inch videodisc.

must take hundreds of video screen dumps to make a hard copy of a map area of interest and, even then, the screens do not immediately splice together because of the overlap areas.

the biggest disadvantage is that, since frames have to be arranged in a checkerboard fashion, there is no way to jump in directions other that north, south, east or west and maintain visual continuity. As an example, the visual discontinuity in viewing a "great circle" route from Alaska to New York would be unbearable for all but the most hearty.

The digital mapping approach has been around for at least 20 years and is much more frequently used than the analog approach. Digital data bases are stored in computers in a format similar to text of other databases. Unlike map information on a videodisc, the outstanding map features are stored as a list of objects to be drawn, each object being defined by a plurality of vector "dot" coordinates which define the crude outline of the object. As one example, a road is drawn by connecting a series of dots which were chosen to define the path (i.e., the "outline") of the road. Once drawn, further data and processing can be used to smooth the crude outline of the object, place text, such as the name or description of the object in a manner similar to what happens when drawing on a paper map.

Advantages of the digital approach:

digital maps are the purest form of geographical mapping data: from them, paper and analog maps can be produced.

digital maps can be quickly and easily updated in near real-time, and this updating can be in response to data input from external sources (e.g., geographical monitoring devices such as satellite photography).

digital maps can be easily modified to effect desirable mapping treatments such as uncluttering, enhancing, coloring, etc.

digital maps can be easily and accurately scaled, rotated and drawn at any perspective view point.

digital maps can be caused to reproduce maps in 3-D.

digital maps can drive pen-plotters (for easy paper reproductions), robots, etc.

digital maps can be stored on any mass storage device.

Disadvantages of the digital approach:

digital maps require the use or creation of a digital database: this is a very time-consuming and expensive process, but once it is made, the data base can be very easily copied and used for many different projects.

The digital approach is utilized with the present invention, as this approach provides overwhelming advantages over the above-described paper and analog approaches.

In designing any mapping system, several features are highly desirable:

First, it is highly desirable that the mapping system be of low cost.

Second, and probably most important, is access time. Not only is it generally desirable that the desired map section be accessible and displayed within a reasonable amount of time, but in some instances, this access time is critical.

In addition to the above, the present invention (as mentioned above), seeks to provide a third important feature,—a mapping system which allows the manipulation of and access to an extraordinary amount of mapping information, i.e., a mapping system which allows a user to quickly and easily access a detailed map of any geographical area of the world.

A tremendous barrier is encountered in any attempt to provide this third feature. In utilizing the digital approach to map a large geographical area in detail

(e.g., the earth), one should be able to appreciate that the storage of mapping data sufficient to accurately define all the geographical features would represent a tremendous data base.

While there have been digital mapping implementations which have successfully been able to manipulate a tremendous data base, these implementations involve tremendous cost (i.e., for the operation and maintenance of massive mainframe computer and data storage facilities). Furthermore, there is much room for improvement in terms of access time as these mainframe implementations result in access times which are only as quick as 20 seconds. Thus, there still exists a need for a low-cost digital mapping system which can allow the storage, manipulation and quick (i.e., "real time") access and visual display of a desired map section from a tremendous mapping data base.

There are several additional mapping system features which are attractive.

It is highly desirable that a mapping system be sensitive to and compensate for distortions caused by mapping curved geographical (i.e., earth) surfaces onto a flat, two-dimensional representation. While prior art approaches have provided numerous methods with varying degrees of success, there is a need for further improvements which are particularly applicable to the digital mapping system of the present invention.

It is additionally attractive for a mapping system to easily allow a user to change his/her "relative viewing position", and that in changing this relative position, the change in the map display should reflect a feeling of continuity. Note that the "relative viewing position should be able to be changed in a number of different ways. First, the mapping system should allow a user to selectively cause the map display to scroll or "fly" along the geographical map to view a different (i.e., "lateral") position of the geographical map while maintaining the same degree of resolution as the starting position. Second, the mapping system should allow a user to selectively vary the size of the geographical area being displayed (i.e., "zoom") while still maintaining an appropriate degree of resolution, i.e., allow a user to selectively zoom to a higher "relative viewing position" to view a larger geographical area with lower resolution regarding geographical, political and cultural characteristics, or zoom to a lower "relative viewing position" to view a smaller geographical area with higher resolution. (Note that maintaining the appropriate amount of resolution is important to avoid map displays which are effectively barren or are cluttered with geographical, political and cultural features.) Again, while prior art approaches have provided numerous methods with varying degrees of success, there is a need for further improvements which are particularly applicable to the digital mapping system of the present invention.

The final feature concerns compatibility with existing mapping formats. As mentioned above, the creation of a digital database is a very tedious, time-consuming and expensive process. Tremendous bodies of mapping data are available from many important mapping authorities, for example, the U.S. Geological Survey (USGS), Defense Mapping Agency (DMA), National Aeronautics and Space Administration (NASA), etc. In terms of both being able to easily utilize the mapping data produced by these agencies, and represent an attractive mapping system to these mapping agencies, it would be highly desirable for a mapping system to be compatible with all of the mapping formats used by these respective agencies. Prior art mapping systems have been deficient in this regard; hence, there still exists a need for such a mapping system.

SUMMARY OF THE INVENTION

The present invention provides a digital mapping method and system of a unique implementation to satisfy the aforementioned needs.

The present invention provides a computer implemented method and system for manipulating and accessing digital mapping data in a tremendous data base, and for the reproduction and display of electronic display maps which are representative of the geographical, political and cultural features of a selected geographical area. The system includes a digital computer, a mass storage device (optical or magnetic), a graphics monitor, a graphics controller, a pointing device, such as a mouse, and a unique approach for structuring, managing, controlling and displaying the digital map data.

The global map generating system organizes the mapping data into a hierarchy of successive magnitudes or levels for presentation of the mapping data with variable resolution, starting from a first or highest magnitude with lowest resolution and progressing to a last or lowest magnitude with highest resolution. The idea of this hierarchical structure can be likened to a pyramid with fewer stones or "tiles" at the top, and where each successive descending horizontal level or magnitude contains four times as many "tiles" as the level or magnitude directly above it. The top or first level of the pyramid contains 4 tiles, the second level contains 16 tiles, the third contains 64 tiles and so on, such that the base of a 16 magnitude or level pyramid would contain 4 to the 16th power or 4,294,967,296 tiles. This total includes "hyperspace" which is later clipped or ignored. Hyperspace is that excess imaginary space left over from mapping of 360 deg, space to a zero magnitude virtual or imaginary space of 512 deg, square.

A first object of the present invention is to provide a digital mapping method and system which are of low cost.

A second and more important object of the present invention is to provide a unique digital mapping method and system which allow access to a display of the geographical, political and cultural features of a selected geographical area within a minimum amount of time.

A third object of the present invention is to provide a digital mapping method and system which allow the manipulation of and access to an extraordinary amount of mapping information, i.e., a mapping method and system which allow a user to quickly and easily access a detailed map of any geographical area of the world.

Another object of the present invention is to provide a digital mapping method and system which recognize and compensate for distortion introduced by the representation of curved (i.e., earth) surfaces onto a flat two-dimensional display.

Still a further object of the present invention is to provide a digital mapping method and system which allow a user to selectively change his/her "relative viewing position", i.e., to cause the display monitor to scroll or "fly" to display a different "lateral" mapping position of the same resolution, and to cause the display monitor to "zoom" to a higher or lower position to display a greater or smaller geographical area, with an appropriate degree of resolution.

A fifth object of the present invention is to provide a digital mapping method and system utilizing a unique

mapping graticule system which allows mapping data to be compatibly adopted from several widely utilized mapping graticule systems.

BRIEF DESCRIPTION OF THE DRAWINGS

The foregoing and other objects, structures and features of the present invention will become more apparent from the following detailed description of the preferred mode for carrying out the invention; in the description to follow, reference will be made to the accompanying drawings in which:

FIG. 1 is an illustration corresponding to a flat projection of the earth's surface.

FIG. 2 is an illustration of a digital computer and mass storage devices which can be utilized in implementing the present invention.

FIGS. 3A–3F are illustrations of monitor displays showing the ability of the present invention to display varying sizes of geographical areas at varying degrees of resolution.

FIG. 4 is a cross-sectional diagram of a simple building example explaining the operation of the present invention.

FIG. 5A and B are plan view representations of a paper 450 as it is viewed from the relative viewing position A shown in FIG. 4.

FIG. 6 is a plan view representation of a paper 450 as it is viewed from the relative viewing position B shown in FIG. 4.

FIG. 7 is a plan view representation of a paper 450 as it is viewed from the relative viewing position C shown in FIG. 4.

FIG. 8 is a pyramidal hierarchy of the data base file structure showing an example of the ancestry which exits between files.

FIG. 9A is a plan view representation of a paper 450, with the paper being divided into a first level of quadrant areas.

FIG. 9B is an illustration of a monitor displaying a digital map of the area enclosed by the dashed portions in FIG. 9A.

FIG. 10A is a plan view representation of a paper 450, with the upper-left and lower-right paper quadrant areas being further divided into quadrants.

FIG. 10B is an illustration of a monitor displaying a digital map of the area enclosed by the upper-left dashed portion in FIG. 10A.

FIG. 11A is a plan view representation of a paper 450, with several sections of the second level of quadrants being further divided into additional quadrants.

FIG. 11B is a higher resolution display of the area enclosed within the dashed portion in FIG. 11A.

FIG. 12 is a plan view illustration of a quadrant area division, with a two-bit naming protocol being assigned to each of the quadrant areas.

FIG. 13 is a pyramidal hierarchy of the data base files using the two-bit naming protocol of FIG. 12, and showing an example of the ancestry which exits between files.

FIG. 14 is a plan view illustration of a 360°×180° flat projection of the earth being impressed in the 512°×512° mapping area of the present invention, with a first quadrant division dividing the mapping area into four equal 250°×256° mapping areas.

FIG. 15 is the same plan view illustration of FIG. 14, with a second quadrant division dividing the mapping area into 16 equal 126°×128° mapping areas.

FIG. 16 is the same plan view illustration of FIG. 15, with a third quadrant division dividing the mapping area into 64 equal 64°×64° mapping areas.

FIG. 17 is the same plan view illustration of FIG. 16, with a fourth quadrant division dividing the mapping area into 256 equal 32°×32° mapping areas.

FIG. 18 is the same plan view illustration of FIG. 17, with a fifth quadrant division dividing the mapping area into 1024 equal 16°×16° mapping areas.

FIG. 19 is the same plan view illustration of FIG. 18, with a sixth quadrant division dividing the mapping area into 4096 equal 8°×8° mapping areas.

FIG. 20 is an illustration showing the application of polar compression at the 8th level or magnitude of resolution.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS OF THE INVENTION

Before turning to the detailed description of the preferred embodiments of the invention, it should be noted that the map illustrations used throughout the drawings are only crude approximations which are only being used to illustrate important features and aspects and the operation of the present invention; therefore. the geographical political and cultural outlines may very well differ from actual outlines.

FIG. 1 is a crude representation of what the earth's surface would look like if it were laid flat and viewed from a "relative viewing position" which is a great distance in space. Shown as vertical lines are: 10, corresponding to the 0° meridian extending through Greenwich, England; 20, corresponding to the 180° west meridian: and, 30, corresponding to the 180° east meridian. Shown as horizontal lines are: 40, corresponding to the equator: 50, corresponding to 90° north (i.e.. the north pole): and 60, corresponding to 90° south (i.e.. the south pole).

Note that at this "relative viewing position", not much detail as to cultural features is seen; i.e., all that is seen is the general outline of the main geographical masses of the continents.

The present invention seeks to provide a low cost and efficient computer-based mapping method and system having a unique approach for arranging and accessing a digital mapping database of unlimited size, i.e., a mapping method and system which can manipulate and access a data base having sufficient data to allow the mapping system to reproduce digital maps of any geographical area with different degrees of resolution. This can be most easily understood by viewing FIG. 2 and FIGS. 3A–F.

Because of the overwhelming advantages over the paper and analog mapping approaches, the digital mapping approach is utilized with the present invention; thus, there is shown in FIG. 2, a digital computer 200, having a disk or hard drive 280, a monitor 210, a keyboard 220 (having a cursor control portion 230), and a mouse device 240. As mentioned previously, in a digital mapping approach, mapping information is stored in a format similar to the text of other databases, i.e., the outstanding map features are stored as a list of objects to be drawn, each object being defined by a plurality of vector "dot" coordinates which define the crude outline of the object. (Note: the reproduction of a digital map from a list of objects and "dot" vectors is well known the art, and is not the subject matter of the present invention; instead, the invention relates to a unique

method and system for storing and accessing the list of objects and "dot" vectors contained in a tremendous digital data base.)

Once a geographical map has been "digitized",—i.e., converted to a list of objects to be drawn and a plurality of vector "dot" coordinates which define the crude outline of the object —, the mapping database must be stored in the memory of a mass storage device. Thus, the digital computer **200**, which is to be used with the mapping method and system of the the present invention, is shown associated with the magnetic disk **260** (which represents any well-known magnetic mass storage medium, e.g., floppy disks, hard disks. magnetic tape, etc.), and the CD-ROM 270 (which represents any well-known optical storage medium, e.g. a laser-read compact disk). Alternatively, the digital mapping database can be stored on, and the digital computer can be associated with any well known electronic mass storage memory medium (e.g., ROM, RAM, etc.). Because of every increasing availability. reductions in cost, and tremendous storage capacities, the preferred memory mass storage medium is the CD-ROM, i.e., a laser-read compact disk.

The discussion now turns to FIGS. **3A–F**, showing illustrations of monitor displays which provide a brief illustration of the operation of the present invention. Although the digital nature of the maps of FIGS. **3A–3F** can easily be detected due to the jagged outlines, it should be understood that these geographical outlines could easily be smoothed using any of a number of "smoothing" techniques which are well-known to those skilled in the digital mapping art.

In FIG. **3A**. the digital computer has retrieved relevant mapping information from the digital mapping database, and has produced a monitor display of a digital map substantially corresponding to the flat projection of the earth's surface which was shown in FIG. **1**. In FIG. **3A**, the monitor display reflects a "relative viewing position" which is a great distance in space, and hence, only the crude geographical outline of the continents is shown with sparse detail.

Suppose a user wishes to view a map of the states of Virginia and Maryland in greater detail. By entering the appropriate commands using the keyboard **220** or the mouse device **240**, a user can cause the monitor display to "zoom" to a lower "relative viewing position", such that the monitor displays a digital map of a smaller geographical area which is shown at a higher degree of resolution. Thus, in FIG. **3B** the a digital map of the continents of the western hemisphere is displayed in greater detail.

By entering additional commands, a user can cause the monitor display to further "zoom" to the following displays: FIG. **3C** showing North America in greater detail; FIG. **3D** showing the eastern half of the United States in greater detail: FIG. **3E** showing the east coast of the United States in greater detail; and. FIG. **3F** showing Virginia and Maryland in greater detail.

Although in this example, the monitor display was caused to "zoom" to Virginia and Maryland, it should, be appreciated that the present invention allowed a user to selectively zoom into any geographical area of the earth, and once a user has reached the desired degree of mapping resolution, the mapping system of the present invention also allows the user to "scroll" or "fly" to a different lateral position on the map.

Furthermore, although the drawings illustrate the monitor display zooming to display state boundaries, and features, it should be further appreciated that the present invention is by no means limited to this degree of resolution. In fact, the degree of resolution capable with the present invention will be shown to be limited only by the operating system of the digital computer **200** with which the present invention is used. In one demonstration, the monitor display has been shown to be able to zoom to resolution where the outlines of streets were displayed. Even further degrees of resolution are possible as will be more fully understood after the discussions below.

In digitally mapping a large geographical area (e.g., the earth) in detail, —especially in the degree of resolution mentioned above —, one should be able to appreciate that the storage of digital mapping data sufficient to accurately define all the geographical, political and cultural features would represent a tremendous digital mapping database. In order to provide a low cost mapping system having quick access time and allowing a high degree of resolution, what is needed is a mapping system having an effective approach for arranging an accessing the digital database. Prior art mapping systems have been deficient in this regard.

The mapping system of the present invention utilizes a new and extremely effective approach, which can be most easily understood using the following simplified example.

In FIG. **4**, there is shown the cross-section of a building **400**, with a square hole **410** (shown in cross-section) cut through the third level floor **420**. with a larger square hole **430** (shown in cross-section) cut in the second level floor **440**, and with a large square piece of paper **450** (shown in cross-section) laid out on the first level floor **460**. Suppose it was desired to build up a digital data base which could be used to reproduce a digital map of the paper **450** with varying degrees of resolution.

First, one would take the "relative viewing position" A, and view the paper **450** through the square hole **410** in the third level floor **420**. At this level. the paper **450** appears small (FIG. **5A**), and the degree of resolution is such that the message appears only as a series of dots. In order to build up a digital mapping database, the visual perception (FIG. **5A**) is imagined to be divided into four equal quadrants a, b, c, d (FIG. **5B**), and visual features appearing in each respective area is digitized and stored in a separate database file. Thus, four separate database files can be utilized to reproduce a digital map of the paper **450** as viewed from position A (FIG. **4**).

In order to digitize and record data corresponding to a second (or higher) degree of resolution, the next "relative viewing position" B (FIG. **4**) is taken to view the paper **450** through the square hole **430**. At this level, the paper **450** appears larger (FIG. **6**), and the degree of resolution is such that the message now appears as a series of lines. At this second level, the map is imagined as being divided into four times as many areas as the first imaginary division, and then, the visual information contained within each area is digitized and stored in a separate database file. Thus, 16 files can be used to reproduce a digital map of the paper **450**, as viewed from the relative viewing position B (FIG. **4**).

In order to digitize and record data corresponding to a third (or higher) degree of resolution. the next "relative viewing position" C (FIG. **4**) is taken to view the paper **450**. At this level, paper **450** now appears larger (FIG. **7**) and has visual features of higher resolution.

The paper **450** is imagined as being divided into four times as many areas as the second imaginary division, and the visual information is digitized and stored. Thus, 64 files could be used to reproduce a digital map of the paper **450**, as viewed from the relative viewing position C (FIG. **4**).

Once digital data has been entered for the above three "relative viewing positions" A, B, C (FIG. **4**), the digital mapping database contains 4+16+64 or 84 files which can be conceptually envisioned as being arranged in a pyramid structure as shown in FIG. **8**. In order to allow a user to selectively display any desired map section at the desired degree of resolution, the digital computer **200** must be able to know which of the 84 files to access such that the appropriate mapping data can be obtained. The present invention accomplishes this by conceptually arranging the files in a pyramidal structure, and assigning a file name to each file which is related both to the file's position and ancestry within the pyramidal structure. This can be more specifically described as follows:

A file's ancestry can be explained using the illustrations of FIGS. **5B**, **6** and **7**. In FIG. **5B**, the paper **450**, as viewed from "relative viewing position" A (FIG. **4**), is subjected to an imaginary division into four quadrants a, b, c, and d. Quadrants a, b, c, d are related to one another in the sense that it takes all four areas to represent the paper **450**: hence quadrants a, b, c, d can be termed as brothers and sisters.

FIG. **6** is an illustration of the paper **450** as it appears from the relative viewing position B (FIG. **4**). with the paper **450** being subjected to an imaginary division into 16 areas. Note that the areas e, f, g, h (FIG. **6**) represent the same area of paper **450** as the quadrant a (FIG. **5B**). In effect, quadrant a has been enlarged (to show a higher degree of resolution) and divided into quadrants e, f, g, h. Thus, it can be said that quadrant a (FIG. **5B**) is the parent, and that quadrants e, f, g, h (FIG. **6**) are brothers and sisters and the offspring of ancestor a. Similar discussions can be made for quadrants b, c and d and the remaining area of FIG. **6**.

FIG. **7** is an illustration of the paper **450** as it appears from the relative viewing position C (FIG. **4**). with the paper **450** being subjected to an imaginary division into 64 areas. In a manner similar to the discussion above, note that areas s, t, w, x (FIG. **7**) represent the same area of paper **450** as the quadrant h (FIG. **6**). In effect, quadrant h has been enlarged (to show a higher degree of resolution) and divided into quadrants s, t, w, x. Thus, it can be said that quadrant a (FIG. **5B**) is the grandparent, quadrant h (FIG. **6**) is the parent, and quadrants s, t, w, x (FIG. **7**) are the brothers and sisters and offspring of ancestors a and h.

As described previously, once FIGS. **5B**, **6** and **7** are subjected to the imaginary divisions, the visual information in each area (or quadrant) is digitized and stored in a separate file. The 84 resulting files can be conceptually envisioned as the pyramidal structure shown in FIG. **8**. In FIG. **8**, dashed lines are utilized to show the lineage of the files just discussed.

FIG. **8** is further exemplary of one file naming operation which can be utilized with the present invention.

At the top of the pyramidal structure (FIG. **8**). each of the four quadrant files is arbitrarily assigned a different character. A, B, C, D, (Note: The characters assigned are not critical with regard to the invention and hence it should be noted that any characters can be assigned, e.g., **0**,**1**,**2**,**3**, etc.)

In moving down one level in the pyramidal structure,, the filenames for each of the respective files on the second level is increased to two characters.

In calculating the filenames, it is convenient to first divide the second level files into groups of four, according to parentage. To maintain a record of ancestry, the ancestor filename of each file is maintained as the first part of the filename. In determining the second part, the naming protocol which was utilized to name the quadrant files of the top level, is also utilized in naming the respective quadrant files on the second level. Thus, parent file A is shown as being related to descendent (i.e., brother and sister) files AA, AB, AC, AD. Similar discussion can be made for the remaining files along these two level.

A similar process can be utilized in providing the unique filenames to the third level files. At this level, the filenames consist of three characters. Again, the ancestor filename of each file would be maintained as a first filename part, in order to maintain a record of ancestry. In the example illustrated (FIG. **8**), parent file AD is shown as being related to descendent (i.e., brother and sister) files ADA, ADB, ADC, ADD. Similar discussions can be made for the remaining files along these two levels, and furthermore, similar discussions can be made each time a pyramidal level is added.

From the above discussion, one should be able to realize that the above-described naming convention is particularly useful in programming a digital computer to move through the pyramidal file structure to access the appropriate data corresponding to varying degrees of resolution. More particularly, one should be able to realize that, since file names increase one character in length each time there is a downward movement through the pyramidal structure and the protocol for naming descendent files is known, the digital computer can be programmed to quickly and easily access the appropriate files for a smaller mapping area with a greater degree of resolution. Similarly, one should be able to realize that, since the filenames decrease one character in length each time there is an upward movement through the pyramidal structure, the digital computer can be programmed to quickly and easily access the appropriate files for a greater mapping area with a smaller degree of resolution.

The following example is believed to provide an increase in the understanding of the present invention.

In the example, it is assumed that the digital database corresponding to the three resolutions of the paper **450** (as shown in FIGS. **4**, **5A–B**, **6**, **7**) have been loaded to be accessible from the memory mass storage device, and furthermore, it is assumed that the mapping system is programmed to initially access and display a digital map corresponding to the digital mapping data in the files A, B, C, D (FIG. **8**). Thus, the monitor (FIG. **9B**) would display (in low resolution) the entire area enclosed within dashed portion **900** illustrated on the paper **450** (FIG. **9A**). (Note: The reproduction of a digital map from digital data from several different files or sources is well-known in the art and is not the subject matter of the present invention.)

Suppose the user notices the dotted area on the low resolution map and wishes to investigate this area further. By using the appropriate keys (e.g. ↖, ↗, ↙, ↘) and/or a mouse device, a user can give the mapping system an indication that he/she wishes to see the smaller area (i.e., quadrant A) at a higher degree of resolution. Upon receiving this preference, the mapping

system can use its knowledge of the file naming operations to quickly determine the names of the files which must be accessed. More specifically, using A as the parent file name and following the existing quadrant naming protocol the mapping system is quickly and easily able to calculate that it is files AA, AB. AC, AD which it needs to access. Once these files are accessed, the monitor in FIG. **10B** displays (in higher resolution) the area enclosed within the dashed portion **1000** as illustrated on the paper **450** (FIG. **10A**).

If a user is still not satisfied with the degree of mapping resolution, the user can again use the appropriate keys or mouse device to indicate that he/she wishes to see the smaller area (e.g., quadrant D; FIG. **10A**) in a higher degree of resolution. In using AD as the parent filename and following the existing quadrant naming protocol, the mapping system is quickly and easily able to calculate that it is files ADA, ADB, ADC, ADD which it needs to access. Once these files are accessed, the monitor (FIG. **11B**) displays (in higher resolution), the area enclosed within the dashed portion **1100** as illustrated on the paper **450** (FIG. **11A**).

One skilled in the digital mapping and computer programming art should recognize that "scrolling" or "flying" to different lateral "relative viewing positions" to display a different lateral portion of the map is also provided by the present invention. Instead of adding or removing filename characters as in a change of resolution, in this instance, the mapping system must be programmed to keep track of the filenames of the current position and also, the orderly arrangement of filenames so that the appropriate filenames corresponding to the desired lateral position can be determined. As an example if the user desired to scroll to the right border of the paper **450**, the mapping system would respond by accessing and causing the monitor to display the digital maps corresponding to the following sequence of files: (Note: In this example, it is assumed that it takes 4 files to provide sufficient digital data to display a full digital map on a monitor) ADA, ADB, ADC, ADD; ADB, ADD, BCA, BCC; BCA, BCB, BCC, BCD; BCB, BCD, BDA, BDC; and BDA, BDB, BDC, BDD. If the user, then desired to scroll to the bottom (right corner) of the paper **450**, the mapping system would respond by accessing and causing the monitor to display the digital maps corresponding to the following files: BDA, BDB, BDC, BDD; BDC, BDD, DBA, DBB; DBA, DBB, DBC. DBD; DBC, DBD, DDH, DDB; DDA, DDB, DDC, DDD. In effect as all of the files in the above example correspond to the same level of resolution all these files (and any group of files which exist on the same level of resolution) can be taken as being related as cousins.

FIGS. **9A**, **10A**, **11A** can also be used to illustrate the operation of moving toward the display of a larger mapping area with a lower degree of resolution.

Assume that after lateral "scrolling" or "flying", that the monitor is now displaying (not shown) a digital map corresponding to the enclosed area **1110** shown in FIG. **11A**. (Note: at this position the mapping system is accessing and display a digital map corresponding to the digital data in the files DCA, DCB, DCC, DCD). Suppose the user now wishes to cause the "relative viewing position" to zoom upward, such that the monitor will display a larger portion of the paper **450** at a lower degree of resolution. By using the appropriate keys or a mouse device, the user indicates his/her preference to the mapping system. Upon receiving this preference, the mapping system is programmed to quickly determine the names of the files which must be accessed. More specifically, the mapping system is able to look at the first portion of the filenames currently being used (i.e., DCA, DCB, DCC, DCD), to immediately determine that these files have the ancestry DC, i.e., have a grandfather D and a parent DC. The mapping system then immediately determines brother and sister files of parent file DC as being DA, DB and DD. The mapping system then accesses these files and causes the monitor to display a digital map (not shown) corresponding to the enclosed portion **1010** (FIG. **10A**) of the paper **450**.

Suppose the user again indicate a preference to cause the "relative viewing position" zoom upward. Upon receiving this preference, the mapping system again goes through a process similar to that discussed immediately above. However, this time the mapping system looks at the filenames currently being used (i.e., DA, DB, DC, DD) and determines that parent file D has brother and sister files A, B and C. The mapping system then immediately accesses these files and causes the monitor to display a digital map (FIG. **9B**) corresponding to the enclosed portion **900** (FIG. **9A**) of the paper **450**.

The text now turns to a description of the operation for assigning unique filenames in the currently preferred embodiment, i.e., in a digital mapping system which is implemented in a DOS operating system.

As anyone skilled in the computer art will know. every computer operating system has its own unique set of rules which must be followed. In an implementation of the present invention in a DOS operating system. the DOS rules must be followed. Since a critical feature of the present invention is the division of the digital mapping database into a plurality of files (each having a unique filename), of particular concern with the present invention is the DOS rules regarding the naming of filenames.

A DOS filename may be up to eight (8) characters long, and furthermore. may contain three (3) additional trailing characters which can represent a file specification. Thus, a valid DOS filename can be represented by the following form:

—

where "–" can be replaced by any ASCII character (including blanks), except for the following ASCII characters:

. " / \ [ ] : ¦ < > + ; ,

and ASCII characters below **20H**. The currently preferred embodiment stays within these DOS filename rules by using the file naming operations which are detailed below.

Because the assigned filenames will be seen to be related to hexadecimals, a useful chart containing the hexadecimal base and also a conversion list (which will be shown to be convenient ahead), is reproduced below:

Column 1	Column 2	Column 3
0000	0	G
0001	1	H
0010	2	I
0011	3	J
0100	4	K
0101	5	L
0110	6	M

-continued

Column 1	Column 2	Column 3
0111	7	N
1000	8	O
1001	9	P
1010	A	Q
1011	B	R
1100	C	S
1101	D	T
1110	E	U
1111	F	V

The first column contains a list of all the possible 4-bit binary combinations: the second column contains the hexadecimal equivalent of these binary numbers: and the third column concerns a "mutant-hex" conversions which will be shown to be important in the discussion to follow. In the operatidns to assign unique filenames for use in a DOS operating system, the present invention looks at each of the eight DOS filename characters as hexadecimal characters rather than ASCII characters. Hence, while the following discussion will center around determining unique filenames using hexadecimal (and "mutant-hexadecimal") characters, it should be understood in an actual DOS implementation, the hexadecimal filenames must be further converted into the equivalent ASCII characters such that the appropriate DOS file naming rules are followed.

At this point, it is also useful to note that the file naming operation of the preferred embodiment is not concerned with the trailing three character filename extension. However, it should be further noted that this three character filename extension may prove useful in specifying data from different sources, and allowing the different types of data to reside in the same database. As examples, the filename extension ".spm" might specify data from scanned paper maps, the filename extension ".si" might specify data from satellite imagery, the filename extension ".ged" might specify gridded elevation data, etc.

As a result of the foregoing and following discussions, it will be seen that the naming operation of the preferred embodiment is concerned only with a filename of the following form:

-

where each "-" represents a character which is a hexadecimal character within the character set of "0–9" and "A–F", or is a "mutant-hexadecimal" character within the character set of "G–V".

Several more important file naming details should be discussed.

First, it should be pointed out that the first four (4) filename characters is designated as corresponding to the "x" coordinate characters, and the last four (4) filename characters are designated as corresponding to the "y" coordinate characters.

Second, during the file naming operations, often it is necessary to convert the filename characters into the equivalent binary representation. As each hexadecimal character can be converted into a four bit binary number, it can be seen that the first four (4) filename characters (designated as "x" coordinate characters) can be converted into sixteen (16) binary bits designated as "x" bits, and similarly, that the last four (4) filename characters (designated as "y" coordinate characters) can be converted into sixteen (16) binary bits designated as "y" bits. As will become more apparent ahead, each of these sixteen (16) "x" and "y" bits corresponds to a filename bit which can be manipulated when assigning filenames at a corresponding magnitude or level of mapping resolution, e.g., the first "x" and first "y" bits correspond to filename bits which can be manipulated when assigning unique filenames at the first magnitude, the second "x" and second "y" bits correspond to filename bits which can be manipulated when assigning unique filenames at the second magnitude, etc.

Third, FIG. **12** corresponds to the naming protocols which are utilized to modify and relate a parent filename to four (4) quadrant filenames. Note that there is a two-bit naming protocol in each of the quadrant files. As will become more clear ahead, the first bit of each protocol determines whether the current "x" filename bit will be modified (i.e., if the first protocol bit is a "1", the current "x" filename bit is changed to a "1", and if first protocol bit is a "0", the current "x" filename bit is maintained as a "0"), and the second bit determines whether the current "y" filename bit will be modified (in a similar manner).

The text now turns to a file naming example which is believed to provide further teachings and clarity to the currently preferred file naming operation.

FIG. **13** is an illustration of a portion of the preferred digital data base, with the plurality of files (partially shown) being arranged in a conceptual pyramidal manner in a manner similar to that which was described with reference to FIG. **8**. More specifically, there are shown four files **1300** having digital data corresponding to a first level or magnitude of mapping resolution, sixteen files **1310** having digital data corresponding to a second level or magnitude of mapping resolution, sixty-four files **1320** having digital data corresponding to a third level or magnitude of mapping resolution, and a partial cut-away of a plurality of files **1330** having data corresponding to a fourth level or magnitude of mapping resolution. Although not shown, it is to be understood that, in the preferred embodiment, additional pyramidal structure corresponding to levles magnitudes five through sixteen similarly exist. As examples of the file naming operation, filenames will now be calculated for the files which essentially occupy the same positions as the files which were outlined in FIG. **8**.

We begin with the initializing eight (8) character filename:

0	0	0	0	0	0	0	0

which can be converted to the binary equivalent:

0000	0000	0000	0000	0000	0000	0000	0000

This binary representation is the basic foundation which will be used to calculate all of the filenames for the files on the first level (**1300**). Note, that the first and last four filename characters, and the first and last sixteen bits are slightly separated in order to conveniently distinguish the "x" and "y" coordinate characters and bits. Both the first (leftmost) "x" bit and the first (leftmost) "y" bit are the bits which can be manipulated in assigning a unique filename to the files on the first level.

File naming begins with the first (upper-rightmost) file on the first level **1300**. The naming protocol assigned to this quadrant file is the two-bit protocol "10".

As the first protocol bit is a "1", this means that the current "x" bit must be changed to a "1". As the second protocol bit is a "0", this means that the current "y" bit is maintained as a "0". As a result of the foregoing, the first (upper-rightmost) file is assigned the filename having the binary equivalent of:

1000	0000	0000	0000	0000	0000	0000	0000

which can be converted to the hex characters:

8	0	0	0	0	0	0	0.

In proceeding clockwise, next is the second (lower-rightmost) file on the first level **1300**. The naming protocol assigned to this quadrant file is the two-bit protocol "11". As the first protocol bit is a "1", the current "x" bit is changed to a "1": similarly, as the second protocol bit is a "1", the current "y" bit is changed to a "1". As a result of the foregoing, the second (lower-rightmost) file is assigned the filename having the binary equivalent of:

1000	0000	0000	0000	1000	0000	0000	0000

which can be converted to the hex characters:

8	0	0	0	8	0	0	0.

Continuing clockwise, next is the third (lower-leftmost) file on the first level **1300**. The naming protocol assigned to this quadrant file is the two-bit protocol "01". As the first protocol bit is a "0", the current "x" bit is maintained at 0. As the second protocol bit is a "1", the current "y" bit is changed to a "1". As a result of the foregoing. the third (lower-leftmost) file is assigned the filename having the binary equivalent of:

0000	0000	0000	0000	1000	0000	0000	0000

which can be converted to the hex characters:

0	0	0	0	8	0	0	0.

Finally, there is the fourth (upper-leftmost) file on the first level **1300**. The naming protocol assigned to this quadrant is the two-bit protocol "00". As neither of the protocol bits is a "1", it can be easily seen that neither of the current "x" and "y" bits changes, and hence, the fourth (upper-leftmost) file is assigned the filename having the binary equivalent of:

0000	0000	0000	0000	0000	0000	0000	0000

which can be converted to the hex characters:

0	0	0	0	0	0	0	0.

In further discussions of the example, it is important to note that the initializing (8) character filename of 0000 0000 (which was utilized to calculate the filenames of the files on the first level 1300) is not utilized in assigning filenames on subsequent levels. In naming files from the second level or magnitude downward, the binary equivalent of the parent file's name is utilized as the foundation from which the descendent file's name is derived. It is only coincidental that the filename of the parent file 00000000 (located in the user-left most corner of the first level **1300**) is the same as the initializing filename. Use of the parent's filename to calculate the descendent's filename will become more readily apparent ahead in the example.

In continuing the file naming example, the fourth (upper-leftmost) file (having filename 00000000) in the first level **1300** can be viewed as being the parent file of the four (highlighted) quadrant files in the second level **1310**. As stated above, the binary equivalent of parent file's 00000000 name is utilized as the foundation for calculating the descendent file's filenames. At this second level or magnitude, the second "x" and "y" bits from the left in the parent's binary filename are taken as the "current" bits which can be manipulated to provide a unique filename for the descendent files.

As the calculation of the filename for the fourth (upper-leftmost) file of the second level **1310** illustrates a very important modification in the file naming operation, the example will first continue with discussions corresponding to this file.

As the naming protocol assigned to the fourth (upper-leftmost) file of the second level **1310** is two-bit protocol "00", it can be seen that neither of the current "x" and "y" bit would be changed. Hence the parent's filename 00000000 is unchanged, and is attempted to be adopted as the descendent's filename. However, note that this is extremely undesirable as the operation of the present invention is based on assigning each data file a unique filename, and furthermore, a DOS operation system will not allow the same filename to be assigned to two different files. To avoid this clash. the preferred file naming operation of the present invention incorprates a further step which can be detailed as follows:

First calculate the filename as explained above. Once the binary filename is obtained, convert to the eight character hexadecimal equivalent.

Next, take the decimal number of the current level or magnitude and subtract one (1) to result in a decimal magnitude modifier. Convert the decimal magnitude modifier into a four-bit binary magnitude modifier, and line these four bits up with the four hexadecimal "x" filename characters. Whenever a "1" appears in the binary magnitude modifier. the corresponding aligned "x" filename character is converted to a "mutant-hexadecimal" character. i.e., a decimal 16 value is added to convert the aligned filename character into a one of the "mutant-hexadecimal" characters in the character set of "G–V".

Conversions from a hexadecimal character to a "mutant-hexadecimal" character can be most readily made using the chart detailed above. As an example, if decimal 16 is added to the hex character "0" (Column 2), there is a conversion to the "mutant-hexadecimal" character "G" (Column 3). Similarly, if decimal 16 is added to the hex character "1" (Column 2), there is a conversion to the "mutant-hexadecimal" character "H" (Column 3). Similar discussion can be made for the remaining hex and "mutant-hexadecimal characters in the chart.

Once correspondingly aligned filename characters are converted to "mutant-hexadecimal", the resultant

eight (8) characters correspond to the file's unique filename.

The above processing will now be applied to the fourth (upper-rightmost) file of the second level **1310** (which was recently discussed above). The resultant binary filename:

0000	0000	0000	0000	0000	0000	0000	0000

is converted to the hex characters:

0	0	0	0	0	0	0	0.

The level or magnitude two (2) minus one (1) results in a decimal magnitude modifier of one (1). The decimal magnitude modifier is converted to the four-bit binary equivalent and is aligned with the "x" filename characters above, as follows:

0	0	0	1

Only the fourth bit of the binary magnitude modifier is a "1", so only the fourth "x" filename character needs to be converted to "mutant-hexadecimal". From the chart, the hexadecimal character "0" is shown to convert to a "mutant-hexadecimal" character "G". Thus. the unique filename which is assigned to the fourth (upper-leftmost) file of the second level **1310**, is:

0	0	0	G	0	0	0	0.

In continuing the example to calculate the filename for the first (upper-right-quadrant) file of the second level **1310**. it can be seen that this file is assigned the two-bit naming protocol "10". The first protocol bit is a "1" which indicates that the current (second from the left) "x" bit of the parent file's binary filename must be changed to a "1", In contrast, the second protocol bit is a "0", which indicates that the current (second from the left) "y" bit is maintained as "0" Thus the parent filename:

0000	0000	0000	0000	0000	0000	0000	0000

is converted to:

0100	0000	0000	0000	0000	0000	0000	0000

which results in the hex characters:

4	0	0	0	0	0	0	0.

The level or magnitude two (2) minus one (1) results in a decimal magnitude modifier of one (1). The decimal magnitude modifier is converted to the four-bit binary equivalent and is aligned with the "x" filename characters above, as follows:

0	0	0	1

Only the fourth bit of the binary magnitude modifier is a "1", so only the fourth "x" filename character needs to be converted to "mutant-hexadecimal". From the chart, the hexadecimal character "0" is shown to convert to a "mutant-hexadecimal" character "G". Thus, the unique filename which is assigned to the first (upper-right-quadrant) file of the second level **1310**, is:

4	0	0	G	0	0	0	0.

Turning now to the second (lower-right-quadrant) file, this file is assigned the two-bit naming protocol "11". The first protocol bit is a "1" which indicates that the current (second from the left) "x" bit of the parent file's binary filename must be changed to a "1", and similarly, the second protocol bit is a "1", which indicates that the current (second from the left) "y" bit of the parent file's binary filename must be changed to a "1" Thus the parent filename:

0000	0000	0000	0000	0000	0000	0000	0000

is converted to:

0100	0000	0000	0000	0100	0000	0000	0000

which results in the hex characters:

4	0	0	0	4	0	0	0.

The level or magnitude two (2) minus one (1) results in a decimal magnitude modifier of one (1). The decimal magnitude modifier is converted to the four-bit binary equivalent and is aligned with the "x" filename characters above, as follows:

0	0	0	1

Only the fourth bit of the binary magnitude modifier is a "1", so only the fourth "x" filename character needs to be converted to "mutant-hexadecimal". From the chart, the hexadecimal character "0" is shown to convert to a "mutant-hexadecimal" character "G". Thus, the unique filename which is assigned to the second (lower-right quadrant) file of the second level **1310**, is:

4	0	0	G	4	0	0	0.

In applying the above operations to the third (lower-left-quadrant) file of the second level **1310**, it can be easily calculated that the resultant filename is:

0	0	0	G	4	0	0	0.

The example of the file naming operation is further extended to the third level or magnitude. as this example is illustrative of both the use of the parent file's binary filename to calculate the descendent's filename, and the removal of "mutant-hexadecimal" conversions before calculating the descendent's filename.

 Unified Patents Exhibit 1005 App'x A-N

In FIG. 13. the third (lower-right-quadrant) file of the second level **1310** is shown as being the parent of the four (4) quadrant files highlighted in the third level or magnitude **1320**.

The discussion centers on the calculation of the unique filename for the second (lower-right-quadrant) file in the third level **1320**. Before the parent filename can be used as the foundation for calculating the descendent's filename. all "mutant-hexadecimal" conversions must be removed. Thus the parent filename:

4	0	0	G	4	0	0	0.

is converted back to:

4	0	0	0	4	0	0	0.

which is further converted to the binary equivalent:

0100	0000	0000	0000	0100	0000	0000	0000

In continuing the calculation, this second (lower-right-quadrant) file is assigned the two-bit naming protocol "11". The first protocol bit is a "1" which indicates that the current (third from the left) "x" bit of the parent file's binary filename must be changed to a "1", and similarly, the second protocol bit is a "1", which indicates that the current (third from the left) "y" bit of the parent file's binary filename must be changed to a "1". Thus the parent filename:

0100	0000	0000	0000	0100	0000	0000	0000

is converted to:

0110	0000	0000	0000	0110	0000	0000	0000

which results in the hex characters:

6	0	0	0	6	0	0	0.

The level or magnitude three (3) minus one (1) results in a decimal magnitude modifier of two (2). The decimal magnitude modifier is converted to the four-bit binary equivalent and is aligned with the "x" filename characters above, as follows:

0	0	1	0

Only the third bit of the binary magnitude modifier is a "1", so only the third "x" filename character needs to be converted to "mutant-hexadecimal". From the chart, the hexadecimal character "0" is shown to convert to a "mutant-hexadecimal" character "G". Thus, the unique filename which is assigned to the second (lower-right-quadrant) file of the third level **1320**, is:

6	0	G	0	6	0	0	0.

The filenames for several additional third level files will be given to give the patent reader further practice.

In applying the above operations to the first (upper-right-quadrant) file of the third level **1320**, it can be easily calculated that the resultant filename is:

6	0	G	0	4	0	0	0.

In applying the above operations to the third (lower-left-quadrant) file of the third level **1320**, it can be easily calculated that the resultant filename is:

4	0	G	0	6	0	0	0.

Finally, in applying the above operations to the fourth (upper-left-quadrant) file of the third level **1320**, it can be easily calculated that the resultant filename is:

4	0	G	0	4	0	0	0.

As a result of all of the foregoing teachings, one skilled in the art should now be able to calculate the filename of any other of the 1.4 billion files which would be required to provide digital maps corresponding to sixteen (16) resolutions of any geographical area on earth. Furthermore, once a file is being accessed, by understanding the rules and operations of the file naming operation one skilled in the are should be able to calculate any other related files, i.e., parent files. and brother/sister/cousin files.

While the unique approach for storing and accessing files in the pyramidal file structure has been particularly pointed out. further discussion is needed as to an additional advantageous feature of the present invention.

As mentioned previously, the creation of a digital database is a very tedious, time consuming and expensive process. Tremendous bodies of mapping data are available from many important mapping authorities, for example, the U.S. Geological Survey (USGS), Defense Mapping Agency (DMA), National Aeronautics and Space Administration (NASA), etc.

The maps and mapping information produced by the above recited agencies, is always based on well established mapping area divisions. As a few examples, the Defense Mapping Agency (DMA) produces maps and mapping information based on the following mapping areas: GNC maps which are $2° \times 2°$: JNC maps which are $1° \times 1°$; ONC maps which are $30' \times 30'$: TPC maps which are $15' \times 15'$; and JOG maps which are $7.5' \times 7.5'$. As a further example, the U.S. Geological Survey (USGS) also produces maps and utilizes mapping information based on $15' \times 15'$ and $7.5' \times 7.5'$.

In terms of both being able to easily utilize the mapping data produced by these agencies, and represent an attractive mapping system to these mapping agencies, it would be highly desirable for the mapping system of the present invention to be compatible with all of the mapping formats used by these respective agencies. Such is not the case when the mapping database is based on a graticule system corresponding to 360°

If one were to apply multiple quadrant divisions to the $360° \times 180°$ flat map projection of the earth (FIG. **1**). one would result in the following mapping area subdivisions:

Level of quadrant div.:	Resultant mapping area:
1	(4) 180° × 90°
2	(16) 90° × 45°
3	(64) 45° × 22.5°
4	(256) 22.5° × 11.25°
5	(1024) 11.25° × 5.625°
etc.	

Note that these mapping area subdivisions are very awkward, and do not match any of the well settled mapping area subdivisions. (It should be further noted that no better results are obtained if the initial map projection is imagined as being a 360°×360° square instead of a rectangle.)

In order to avoid these awkward mapping subdivisions, and result in quadrant divisions which precisely match widely used mapping area subdivisions, the present invention utilizes a unique initial map projection.

More specifically, as can be seen in FIG. **14**, the present invention initially begins with a unique 512°×512° initial map projection. Shown centered in the 512°×512° map projection is the now familiar 360°×180° flat projection of the surface of the earth. Although the 512°×512° projection initially appears awkward and a waste of map projection space, the great advantages which are resultant from the use of this projection will become more apparent in the discussions to follow.

To aid in this discussion, provided on the next page is a chart which details these important advantages as well as other useful information regarding the use of this map projection.

less complicated, the non-DOS file naming operation will be used in the discussion.

The digital mapping of the earth surfaces begins in FIG. **14**. The visual perception of the earth surfaces is experienced as being centered, and occupying only a portion of the 512°×512° projection. A first quadrant division is applied to result in four equal 256°×256° mapping areas. The visual information in each of the areas is digitized, and stored in a separate file, Thus, it can be seen that one would have to access four files a, b, c, d in order to reproduce a digital map corresponding to the earth surfaces as viewed from this "relative viewing position."

One skilled in the art, might, at this point, wonder if the massive blank portions of the 512°×512° projections result in large blank portions on the digital map display. The preferred embodiment avoid this phenomena, through a simple watchdog operation, i.e., the computer is programmed to keep track of longitudinal and latitudinal movements from an initial position of 0° longitude and 0° latitude, and the computer does not allow scrolling of the monitor display beyond 90° north or south.

As to side to side movements, the computer allows scrolling beyond 180° east or west by patching the appropriate data files together to perform a "wrap around" operation. Note that, with the knowledge of the logical file naming operation, the computer can quickly and easily calculate the appropriate files to access.

Before moving to the next level or magnitude of mapping resolution, it is beneficial to note the correspondence between our findings and the enties in the above-indicated chart.

MAGNITUDE EQUIVALENCY CHART FOR DELORME PROJECTION
Chart assumes 69 statute miles per degree at equator

MAGNITUDE	Window Size without overlap	Ht of window statute miles	Ht of window kilometers	# Windows per MAG	# Windows/MAG w/polar compression	Pixel resolution 480 monitor (ft)	Data resolution (ft) 1024-based window	Equivalent Paper Map Scales	Size of paper map image at equator (in)
1	256° × 256°	17664	28421	4	4		91080		
2	128° × 128°	8832	14211	8	8		45540		
3	64° × 64°	4416	7105	24	24	48576	22770	1:100 million	2.8 × 2.8
4	32° × 32°	2208	3553	72	72	24288	11385	1:50 million	2.8 × 2.8
5	16° × 16°	1104	1776	288	288	12144	5693	1:30 million	2.3 × 2.3
6	8° × 8°	552	888	1152	858	6072	2846	1:16 million	2.2 × 2.2
7	4° × 4°	276	444	4232	3432	3036	1423	1:10 million	1.7 × 1.7
8	2° × 2°	138	222	16200	12808	1518	712	1:5 million	1.7 × 1.7
9	1° × 1°	69	111	64800	51210	759	356	1:2 million	2.2 × 2.2
10	30′ × 30′	34.5	55.5	259000	204840	380	178	1:1 million	2.2 × 2.2
11	15′ × 15′	17.25	27.8	1036800	813600	190	89	1:500,000	2.2 × 2.2
12	7.5′ × 7.5′	8.625	13.9	4147200	3277440	95	44	1:250,000	2.2 × 2.2
13	3.75′ × 3.75′	4.312	6.9	16588800	13109760	47.4	22	1:125,000	2.2 × 2.2
								1:100,000	2.73 × 2.73
								1:80,000	3.4 × 3.4
14	1.875′ × 1.875′	2.156	3.5	66355200	52439040	23.7	11.1	1:62,500	2.2 × 2.2
								1:50,000	2.73 × 2.73
								1:40,000	3.4 × 3.4
15	0.9375′ × 0.9375′	1.078	1.7	265420800	209756160	11.9	5.6	1:24,000	2.8 × 2.8
								1:20,000	3.4 × 3.4
16	0.46875′ × 0.46875′	0.539	0.9	1016683200	839024640	5.9	2.8	1:12,000	2.8 × 2.8

The best way to see the advantages of the 512°×512° mapping projection, is to use it with the previously, taught, quadrant division and pyrimidal file structure to show how this unique mapping projection can provide digital maps of any geographical areas of the earth, with 16 levels or magnitudes of resolution. As it is slightly

In looking at the left-most column, and tracing down to magnitude **1**, note that the 256°×256° window size exactly matches our determination. Furthermore, note that our findings is also in agreement with the number of widows i.e., 4. It is also interesting to note from the third column, that the height or "relative viewing posi-

tion" of this magnitude or level would be 17, 664 statute miles above the earth's surface.

Turning now to the second level or magnitude of resolution (FIG. **15**). a further quadrant division is applied, resulting in sixteen (**16**) mapping areas of 128°×128°. The respective filenames which are assigned to each of the mapping areas is shown. In viewing FIG. **15**, note that there are eight (8) mapping areas which are not intersected by the earth's surface. In order to save valuable memory space, the preferred embodiment will ignore, and in fact will never create these files. Note that there is no use for these files as they do not contain any digital mapping data nor will they ever have any descendents which hold mapping data. In order to implement this "file selectivity", the preferred embodiment again utilizes a watchdog approach. More specifically, as the computer already knows the degree (°) size of the earth's surface and the degree (°) size of each of the mapping areas (i.e., at each level or magnitude of resolution), it can be seen that the computer can easily calculate the filenames which will not intersect the earth's surface.

Again it is useful to correspond our findings with the entries in the chart.

Our findings are substantiated, as, at a magnitude of 2, the window size is shown as being 128°×128°, and there are shown to be eight (8) pertinent windows or files at this magnitude. Again, it is interesting to note that the height or "relative viewing position" of this window would be 8,832 statute miles above the earths earth's surface.

It is important to note that, although the "relative viewing position" of each level or magnitude is moving closer to the earth, the visual perception of the earth (as seen in FIGS. **14–19** is not illustrated as getting larger with a greater degree of detail. This is because or the paper size limitations.

In the third level or magnitude of resolution (FIG. **16**). a further quadrant division is applied, resulting in sixty-four (64) mapping areas of of 64°×64°. As the projection is beginning to represent a large plurality of mapping areas, the filenames have been ommitted. However, it should be understood that the filename assigned to a respective file in this and subsequent degrees of resolution, can easily be calculated by following the previously described file naming operation. In this projection, it can be seen that 40 mapping areas or files are not used, resulting in 24 files which contain the digital mapping data of this resolution. Note that the observed window, and used files again correlates to the entries in the chart. Furthermore, it can be seen that the height or "relative viewing position" is at 4,416 statute miles above the earth.

Further quadrant divisions and the corresponding data can be seen in the FIGS. **17–19** and the chart. From the foregoing discussions, prior teachings, and data from the chart, one skilled in the art should be able to quickly appreciate that a mapping system can be constructed which can provide digital maps corresponding to a plurality of resolutions, of any geographical area of the world.

The chart can now be used to observe the tremendous advantage provided by the 512°×512° projection. In the second column of the chart, one can view the sizes of the mapping area divisions which are produced as a result of the continued quadrant division of the 512°×512° projection. One skilled in the mapping art will be able to fully appreciate that the resultant mapping area divisions exactly correspond to well settled and widely used mapping area formats.

Having described all of the important operations of the present invention, the following further conclusions, comments and teachings can be made.

With the mapping system of the present invention, the mapping data are structured at each magnitude or level into windows, frames or tiles representing subdivisions or partitions of the surface area at the specified magnitude. The windows, frames or tiles of all magnitudes for whatever resolution are structured to receive substantially the same amount or quantity of mapping data for segmented visual presentation of the mapping data by window.

As a further improvement, the lapping system of the present invention can further store and organize mapping data into attributed or coded geographical and cultural features according to the classification and level or resolution or magnitude for presentation on the map display. Several examples of this was previously discussed with regard to the use of the filename extension. If this further improvement is used, the computer can be programmed and arranged for managing and accessing the mapping data, and excluding or including coded features in tiles of a particular magnitude according to the resolution and density of mapping data appropriate to the particular magnitude of the window. The selective display of attributed geographical and cultural features according to resolution maintains or limits the mapping data entered in each tile to no greater than a specified full complement of mapping data for whatever magnitude.

In reviewing the file naming operations which were described, one can see that the global map generating system data base structure relates tiles of the same magnitude by tile position coordinates that are keyed to the control corner of each tile and maintained in the name of the "tile-file". Continuity of same scale tiles is maintained during scrolling between adjacent or neighboring tiles in any direction. The new data base structure also relates tiles of different magnitudes by vertical lineage through successive magnitudes. Each tile of a higher magnitude and lower resolution is an "ancestor tile" encompassing a lineage of "descendant tiles" of lower magnitude and high resolution in the next lower magnitude. Thus the present invention permits accessing, displaying and presenting the structured mapping data by tile, by scrolling between adjacent or neighboring tiles of different magnitude in the same vertical lineage for varying the resolution.

In its simplest form the coordinate system is Cartesian, but the invention contemplates a variety of virtual tile manifestations of windowing the mapping data at each magnitude: for example: tilting the axes; scaling one axis relative to another; having one or both axes logarithmic; or rendering the coordinate space as non-Euclidean all together.

When dealing with vector or point information and gridded data, the most common method is to describe individual points as an x-y offset from the control corner of the tile. In this way the mapping data exist as pre-processed relative points on a spherical surface in a de-projected space. The mapping data can then be projected at the user interface with an application program. When projected, all data ultimately represent points of latitude and longitude. Tiles may also contain mapping data as variable offsets of arc in the x and y directions. The tile header may carry an internal descriptor defin-

ing what type of mapping data is contained. The application or display program may then decode and project the data to the appropriate latitude or longitude positions.

The map generating system contemplates storing analog mapping data in electronic mapping frames in which the raw analog data would be scanned and converted digitally to the tile structure and then later accessed and projected for the purpose of displaying continuous analog mapping data.

In the preferred example embodiment, the digital mapping data are structured by window or tile in a substantially rectangular configuration encompassing defined widths and heights in degrees of latitude and longitude for each magnitude. The mapping data representing each magnitude or level are stored in a deprojected format according to mapping on an imaginary cylindrical surface. For display of the maps, however, the data base manager accesses and presents the tiles in a projected form, according to the real configuration of the mapped surface, by varying the aspect ratio of latitude to longitude dimensions of the tiles according to the absolute position of the window on the surface area.

For example, for a spherical or spheroidal globe having an equator and poles, such as the earth, the mapping data are accessed and displayed by aspecting or narrowing the width in the west-east dimension of the tiles of the same magnitude, while scrolling from the equator to the poles. This is accomplished by altering the width of the tile relative to the height. In the graphics display of each window or tile on the monitor, the tiles are presented essentially as rectangles having an aspect ratio substantially equal to the center latitude encompassed by the tile. Thus, the width of the visual display windows is corrected in two respects. First, the overall width is corrected by aspecting to a narrower width, during scrolling in the direction of the poles, and to a wider width during scrolling in the direction of the equator. Second, the width of the tile is averaged to the center latitude width encompassed by the tile throughout the tile height to conserve the rectangular configuration. Alternatively, or in addition, further compensation may be provided by increasing the number of degrees of longitude encompassed by the tiles during scrolling from the equator to the poles to compensate for the compound curvature of the globe.

A feature and advantage of this new method and new system of map projection are that the dramatic and perverse distortion of the globe near the poles, introduced by the traditional and conventional Mercator projection is substantially eliminated. According to the invention, the compensating aspect ratio of latitudinal to longitudinal dimension of aspecting is a function of the distance from the equator, where the aspect ratio is one, to the poles where the aspect ratio approaches zero, all as described for example in Elements of Cartography, 4th edition. John Wiley & Sons (1978) by Arthur Robinson, Randall Sale and Joel Morrison.

The new system contemplates "polar compression" (FIG. **20**) in the following manner. Starting at 64 degrees latitude, the width of each tile doubles for every eight degrees of latitude. From 72 degrees to 80 degrees latitude, there are 4 degrees of longitude for 1 degree of latitude. From 80 degrees to 88 degrees latitude, it becomes eight to one, and from 88 degrees to the pole (90 degrees) it becomes 16 to one (see illustration of polar compression). (FIG. **20**)

Another feature and advantage of the way in which the new map system and new projection handle polar mapping data are in the speed required to access and display polar data. The new polar compression method drastically minimizes tile or window seeks and standard I/O time. Also, without compressing the poles, the Creation/Edit Software would have to work on increasingly narrow tiles as the aspect ratio approached zero at the poles.

The invention embodies an entirely new cartographic organization for an automated atlas of the earth or other generally spherical or spheroidal globe with 360 degrees of longitude and 180 degrees of latitude, an equator and poles. The digital mapping data for the earth is structured on an imaginary surface space having 512 degrees of latitude and longitude. The imaginary 512 degree square surface represents the zero magnitude or root node at the highest level above the earth for a hierarchial type quadtree data base structure. In fact, the 512 degree square plane at the zero magnitude encompasses the entire earth in a single tile. The map of the earth, of course, fills only a portion of the root node window of 512 degrees square, and the remainder may be deemed imaginary space or "hyperspace".

In the preferred example embodiment from a zero magnitude virtual or imaginary space 512 degrees square, the data base structure of the global map generating system descends to a first magnitude of mapping data in four tiles, windows or quadrants, each comprising 256 degrees of latitude and longitude. Each quadrant represents mapping data for one-quarter of the earth thereby mapping 180 degrees of longitude and 90 degrees of latitude in the imaginary surface of the tile or frame comprising 256 degrees square, leaving excess imaginary space or "hyperspace". In the second magnitude, the digital mapping data are virtually mapped and stored in an organization of 16 tiles or windows each comprising 128 degrees of latitude and longitude.

The map generating system supports two windowing formats, one based on the binary system of the 512 degree square zero magnitude root node with hyperspace and the other based on a system of a 360 degree square root node without hyperspace. A feature and advantage of the virtual 512 degree data base structure with hyperspace are that the tiles or windows to be displayed at respective magnitudes are consistent with conventional mapping scale divisions, for example, those followed by the U.S. Geological Survey (USGS). Defense Mapping Agency (DMA). National Aeronautics and Space Administration (NASA) and other government mapping agencies. Thus, typical mapping scale divisions of the USGS and military mapping agencies include scale divisions in the same range of **1** deg, 30 minutes. 15 minutes. 7.5 minutes of arc on the earth s surface. This common subdivision of mapping space does not exist in a data structure based on a 360 degree model without hyperspace (see chart).

Thus, according to the present invention, the world is represented in an assemblage of magnitudes, with each magnitude divided into adjacent tiles or windows on a virtual or imaginary two-dimensional plane or cylinder. At higher magnitudes the quadtree tiles of mapping data do not fill the imaginary projection space. However, from the seventh magnitude down, the mapping data fills a virtual closed cylinder, and no hyperspace exists at these levels.

In the preferred example embodiment the invention (running on a 16 bit computer) has sixteen magnitudes

or levels (with extensions to 20 levels) representing sixteen altitudes or distances above the surface of the earth. At the lowest (16th) magnitude of highest resolution and closest to the earth, the data base structure contains over one billion tiles or windows (excluding hyperspace), each encompassing a tile height of approximately one half statute mile. At this level of resolution, one pixel on a monitor of 480 pixels in height represents approximately 6 feet on the ground. Mapping data are positioned within each tile using a 0 to 1023 offset coordinate structure, resulting in a data resolution of approximately 3 feet at this level of magnitude (see chart). The contemplated 20th magnitude tile or window height is approximately 175 feet, which results in a pixel resolution of about 4 inches on a monitor of 480 pixels in height and a data resolution of about 2 inches, when utilizing the 0 to 1023 offset coordinate structure. Alternatively, the map-generating system contemplates an extended offset from 10 bits (0 to 1023) to an offset of 16 bits (0 to 65,535). In this case, the extended 20th magnitude results in a data resolution of 3 hundredths of an inch.

For still more resolution, the map generating system contemplates 32 magnitudes on a 32 bit computer and representing 32 altitudes or distances about the surface of the earth. Each level of magnitude may define mapping data within each tile using a 32 bit offset coordinate structure, thereby giving relative mathematical accuracy to a billionth of an inch. In all practicality, 20 separate magnitudes or levels are more than sufficient to carry the necessary levels of resolution and accuracy.

The new invention provides users with the ability graphically to view mapping data from any part of the world-wide data base graphically on a monitor, either by entering coordinates and a level of zoom (or magnitude) on the keyboard, or by "flying" to that location in the "step-zoom" mode using consecutive clicks of the mouse or other pointing device. Once a location has been chosen (this point becomes the user-defined screen center). the mapping software accesses all adjacent tiles needed to fill the entire view window of the monitor and, then, projects the data to the screen. Same scale scrolling is accomplished by simply choosing a new screen center and maintaining the same magnitude.

Vertical zooming up or down is accomplished by choosing another magnitude or level from the menu area with the pointing device or by directly entering location and magnitude on the keyboard. An advantage of this vertical lineage of tiles organized in a quadtree structure is that it affords the efficient and easily followed zooming continuity inherent in the present invention. Further discussion of such quadtree data organization is found in the article. "The Quadtree and Related Hierarchical Data Structures", by Hannan Samet, Computer Surveys. Volume 16 , No. 2, (June 1984), Pages 187 et seq.

The map-generating system also supports many types of descriptive information such as that contained in tabular or relational data bases. This descriptive information can be linked to the mapping data with a latitude and longitude coordinate position but may need to be displayed in alternate ways. Descriptive information is better suited for storage in a relational format and can be linked to the map with a "spatial hook".

In summary, the present invention provides a new automated world atlas and global map generating system having a multi-level hierarchial quadtree data base structure and a data base manager or controller which permits scrolling, through mapping tiles or windows of a particular magnitude, and zooming between magnitudes for varying resolution. While the data base organization is hierarchial between levels or magnitudes, it is relational within each level, resulting in a three dimensional network of mapping and descriptive information. The present invention also provides a new mapping projection that has similarities to the Mercator projection but eliminates drastic distortions near the poles for the purpose of presentation through a method of "aspecting" tile widths as a function of the latitudinal distance from the equator.

While the invention has been particularly shown and described with reference to the preferred embodiment thereof, it will be understood by those skilled in the art that various changes in form and details of the device and the method may be made therein without departing from the spirit and scope of the invention.

What is claimed is:

1. A computer implemented method for generating, displaying and presenting an electronic map from digital mapping data for a surface area having geographical and cultural features, said method comprising the steps of:

organizing the mapping data into a hierarchy of a plurality of successive magnitudes or levels for presentation of said mapping data with variable degrees of mapping resolution, each magnitude for presentation of said mapping data with a different degree of mapping resolution from a first or highest magnitude with lowest resolution to a last or lowest magnitude with highest resolution;

structuring said mapping data at each magnitude into a plurality of windows, frames or files representing subdivisions or partitions of said surface area, said windows of a respective magnitude including mapping data which are appropriate to a degree of mapping resolution being afforded at said magnitude while excluding mapping data which are not appropriate to said degree of mapping resolution, and at least a portion of said windows of each magnitude being structured to receive substantially a same predetermined amount or quantity of mapping data for segmented presentation of the mapping data by window;

organizing said mapping data into records of geographical or cultural features for presentation within said windows, and coding said features;

managing said mapping data for each window by excluding or including coded features appropriate to the degree of mapping resolution and density being afforded by said window, such that a quantity of mapping data entered in each window is no greater than said predetermined amount;

relating windows of a same magnitude by window position coordinates or names and structuring said windows with overlap or mapping data between adjacent or neighboring windows of a magnitude or achieve display continuity during generation, display and presentation of an electronic map;

relating windows of different magnitude by vertical lineage through successive magnitudes, each window of a higher magnitude and lower resolution being an ancestor window being related to a plurality of descendant windows of lower magnitude and higher resolution in a next lower magnitude;

accessing and displaying or presenting mapping data for different positions of a selected magnitude by

scrolling between adjacent or neighboring windows of a same magnitude in predetermined north, south, each and west directions;

and accessing and displaying or presenting mapping data for different selected magnitudes having different resolutions by zooming between windows of different magnitudes in a same vertical lineage.

2. The method of claim 1 further comprising:

organizing said mapping data of said surface area by degrees of latitude and longitude;

structuring each said window of mapping data to represent a substantially rectangular surface area configuration encompassing defined degrees of latitude and longitude for each magnitude, and storing the mapping data for each magnitude in a vertical Mercator projection format;

accessing and presenting said windows of mapping data in a corrected or compensated projection format departing from said Mercator projection format according to a real configuration of said surface area, by varying an aspect ratio of latitude to longitudinal dimensions of each window according to a coordinate position of said window with respect to a coordinate layout of said surface area.

3. The method of claim 2 wherein said surface area comprises a spherical or spheroidal globe having an equator and poles, said method comprising the further steps of:

accessing and presenting mapping data in a corrected projection format by aspecting or narrowing, in a direction from an equator to pole, the width or latitudinal dimension of windows, of a same magnitude, which encompass the same number of degrees of latitude and longitude;

and periodically increasing a number of degrees of longitude encompassed by said windows in said direction from equator to pole to compensate for compound curvature of said globe.

4. The method of claim 1 wherein said surface area comprises a generally spherical or spheroidal globe with 360 degrees of longitudinal, 180 degrees of latitude and an equator and poles, said method comprising the further steps of:

relating windows of different magnitudes by vertical lineage in a hierarchical quadtree database structure, by successively partitioning or subdividing ancestor windows of a vertical lineage into four descent windows or quadrants at a next lower magnitude or level, and incorporating additional records of features in said descendant windows to incorporate mapping data for a next higher resolution.

5. The method of claim 4 wherein said hierarchical quadtree database structure comprises at least sixteen degrees of magnitudes or levels.

6. The method of claim 4 comprising the further steps of:

mapping and storing mapping data for said globe in a virtual Mercator projection format representing an imaginary surface having 512 degrees of longitude and latitude comprising a zero magnitude or root node of said hierarchical quadtree database structure;

mapping and storing a first degree or highest magnitude of mapping data in four windows or quadrants each comprising 256 degrees of longitude and latitude, each window of said first degree of magnitude comprising mapping data for one quarter of said globe thereby mapping 180 degrees of surface area longitude and 90 degrees of surface area latitude in said imaginary surface of 256 degrees of longitude and latitude and leaving excess imaginary space;

mapping and storing a second degree of magnitude of mapping data in sixteen windows each comprising 128 degrees of longitude and latitude of said imaginary surface, each window of said second degree of magnitude comprising mapping data for a further subdivision or partition of said globe;

and mapping and storing third through twelfth degrees of magnitude thereby forming additional levels of a hierarchical quadtree database structure so that an eleventh magnitude comprises windows encompassing 15 seconds of latitude and a twelfth magnitude comprises windows encompassing seven and a half seconds of latitude;

whereby, as a result of the foregoing, windows of said electronic map at respective magnitudes or levels are consistent with conventional mapping scale divisions.

7. The method of claim 6 wherein said hierarchical quadtree database structure comprises sixteen degree of magnitudes or levels including a sixteenth magnitude comprising over 1.4 billion windows, each encompassing approximately a fraction of a minute of a degree of latitude.

8. The method of claim 6 wherein each said window corresponds to a trapezoidal surface area configuration.

9. The method of claim 6 comprising the step of floating mapping data records of selected features from a window of one magnitude to a window of the same vertical lineage in another magnitude.

10. The method of claim 6 comprising the further steps of: generating analog mapping data, structuring said analog mapping data according to a same format as digital mapping data, and overlaying and presenting said digital mapping data and analog mapping data during generation, display and presentation of an electronic map.

11. The method of claim 6 comprising the further step of selectively filling said windows with mapping data so that some windows contain a full complement of mapping data appropriate to a degree of mapping resolution being afforded at said magnitude, and other windows, each of which correspond to a subdivision of surface area containing few or no geographical or cultural features, contain less than a full complement of mapping data.

12. The method of claim 6 comprising the further steps of:

accessing and presenting mapping data in a corrected projection format by aspecting or narrowing, in a direction from an equator to pole, a width or latitudinal dimension of windows, of a same magnitude, which encompass the same number of degrees of latitude and longitude;

and periodically increasing a number of degrees of longitude encompassed by said windows in said direction from equator to pole to compensate for a compound curvature of said globe.

13. The method of claim 12 comprising the further steps of accessing and presenting mapping data in corrected projection format, with each window having a width substantially equal to a center latitude width of said window throughout said window, so that said window is of rectangular configuration.

14. An electronic map generating system including a digital computer, a mass storage device, a display monitor, graphics controller, and system software for structuring, managing, controlling and displaying digital mapping data for a surface area having cultural and geographical features, said system comprising:

a database structure comprising a hierarchical database structure programmed and arranged for organizing said digital mapping data into a hierarchy of a plurality of successive magnitudes or levels for presentation of mapping data with variable resolution, each magnitude for presentation of said mapping data with a different degree of mapping resolution from a first or highest magnitude of lowest resolution to a last or lowest magnitude of lowest resolution to a last or lowest magnitude of highest resolution, and for structuring said digital mapping data at each magnitude into a plurality of windows, frames or files representing subdivisions or partitions of said surface area, said windows of a respective magnitude including mapping data which are appropriate to a degree of mapping resolution being afforded at said magnitude while excluding mapping data which are not appropriate to said degree of mapping resolution, at least a portion of said windows of all magnitudes being structured to receive substantially a same predetermined amount of mapping data for segmented presentation of said mapping data by window, said mapping data being organized into coded records of geographical and cultural features within each window;

a database manager or controller programmed and arranged for managing said mapping data by magnitude or level by excluding or including coded records of features in each window of a particular magnitude according to a resolution and density of mapping data appropriate to the particular magnitude of said each window, and maintaining a quantity of mapping data entered in each window to no greater than a specified full complement whatever the magnitude of the window;

said database structure being programmed to relate windows of a same magnitude by position coordinates or names, and to structure windows of a same magnitude with overlap of mapping data between adjacent or neighboring windows of a magnitude to achieve display continuity during generation, display and presentation of an electronic map, and to relate windows of different magnitude by vertical lineage through successive magnitudes, each window of a higher magnitude and lower resolution being an ancestor window of a plurality of descendant windows of lower magnitude and higher resolution in a next lower magnitude;

said database manager being programmed to access and display or present mapping data for different positions of a selected magnitude by scrolling between adjacent or neighboring windows of a same magnitude in predetermined north, south, east and west directions, and being programmed to access and display or present mapping data for different magnitudes having different resolutions by zooming between windows of different magnitudes in a same vertical lineage.

15. The system of claim 14 wherein said hierarchical database structure is programmed to organize said mapping data by degrees of latitude and longitude and to structure each window of mapping data to represent a substantially rectangular surface area configuration encompassing predetermined degrees of latitude and longitude, said windows for each magnitude being stored in virtual Mercator projection format, said database manager being programmed to access and present windows of mapping data in a corrected or compensated projection format departing from Mercator projection format according to a real configuration of said surface area by varying an aspect ratio of latitude and longitude dimensions of each window according to a coordinate position of said each window with respect to a coordinate layout of said surface area.

16. The system of claim 15 wherein said surface area comprises a spherical or spheroidal globe having an equator and poles, and wherein said database manager is programmed to access and present mapping data in a corrected projection format by aspecting or narrowing, in a direction from an equator to pole, the width or latitudinal dimension of windows, of a same magnitude, which encompass the same number of degrees of longitude, said database manager being further programmed to periodically increase a number of degrees of longitude encompassed by said windows in said direction from equator to pole to compensate for compound curvature of said globe.

17. The system of claim 16 wherein said hierarchical database structure comprises a hierarchical quadtree database structure successively partitioning or subdividing ancestor windows of a vertical lineage into four descendant windows or quadrants at a next lower magnitude or level, and incorporating additional coded records of features in said descendant windows to incorporate mapping data for a next higher resolution.

18. The system of claim 17 wherein said database structure is programmed and arranged to store the mapping data in a virtual Mercator projection representing an imaginary surface having 512 degrees of longitude and latitude comprising a zero magnitude or root node of said hierarchical quadtree database structure, wherein a first degree or first magnitude of mapping data comprises four windows, each window of said first magnitude comprising mapping data for one quarter of said globe on an imaginary surface area of 256 degrees of longitude and latitude, said hierarchical quadtree database structure comprising, in addition to first through tenth magnitudes each having windows which are predetermined subdivisions of said imaginary surface having 512 degrees of longitude and latitude, at least an eleventh magnitude having windows encompassing 15 minutes of latitude, and a twelfth magnitude having windows encompassing 7.5 minutes of latitude, so that windows of a resultant electronic map at respective said eleventh and twelfth magnitudes or levels are consistent with conventional mapping scale divisions.

19. The system of claim 18 wherein said hierarchical quadtree database structure comprises at least 16 degrees of magnitudes or levels, said sixteenth magnitude comprising over 1.4 billion windows, each encompassing degrees of latitude of approximately a fraction of a second of a degree.

20. The system of claim 19 further comprising a database of digital mapping data selectively entered in said database structure, such that some of said windows contain a full complement of mapping data appropriate to a degree of mapping resolution being afforded at said magnitude, and other windows, each of which correspond to a subdivision of surface area containing few or

no geographical or cultural features, contain less than a full complement of mapping data.

21. The system of claim **19** further comprising a database of analog data structured according to a same format as said digital data, and means for overlaying said digital and analog data for electronic map presentation.

22. An electronic map generating system for generating reproductions of a map with selectable degrees of mapping resolution, said map generating system comprising:

database means storing a plurality of computer files containing mapping data corresponding to respective surface areas of a mapping surface, wherein said plurality of computer files is organized into a plurality of successive magnitudes, each magnitude for presentation of said mapping data with a different degree of mapping resolution from a first or highest magnitude with lowest resolution to a last or lowest magnitude with highest resolution, files of a respective magnitude including mapping data which are appropriate to a degree of mapping resolution being afforded at said respective magnitude while excluding mapping data which are not appropriate to said degree of mapping resolution, and wherein a predetermined file naming procedure is utilized to assign, to each respective computer file, a unique filename which:

relates said respective computer file to all other computer files having mapping data corresponding to a same magnitude or degree of mapping resolution; and

relates said respective computer file to any computer file comprising mapping data corresponding to a same surface area of a mapping surface as said respective computer file; and

database manager means for accessing said plurality of computer files using said predetermined file naming procedure, to generate a reproduction of a selected area of a map at a selected degree of mapping resolution.

23. An electronic map generating system as claimed in claim **22**,

wherein each said unique filename is represented by a value contained in a plurality of bits, and

wherein said predetermined file naming procedure:

utilizes a first predetermined subset of said plurality of bits to relate said respective files having mapping data corresponding to a same magnitude or degree of mapping resolution; and

utilizes a second predetermined subset of said plurality of bits to relate said respective computer file to any computer file comprising mapping data corresponding to a same surface area of a mapping surface as said respective computer file.

24. An electronic map generating system as claimed in claim **23**, wherein said unique filename also includes geographical information which can be used to relate a geographical coordinate position of a respective computer file with respect to a coordinate layout of surface areas of said mapping surface.

25. An electronic map generating system as claimed in claim **22**,

wherein an assignment of said unique filenames using said predetermined file naming procedure results in said respective computer files of said plurality to be related in a quadtree database structure.

26. An electronic map generating system as claimed in claim **25**, wherein the respective area of a mapping surface covered within the computer files of consecutive magnitudes or degrees of mapping resolution changes at a predetermined rate in that, when a computer file at a reference magnitude or degree of mapping resolution contains mapping data corresponding to an N×N area of a mapping surface (where N is a real number, and is associated with one of the conventional degree °, minute ′, or second ″ mapping scale divisions), then a computer file at a next consecutive magnitude having a higher degree of mapping resolution contains mapping data corresponding to an (N/2)×(N/2) area of said mapping surface.

27. An electronic map generating system as claimed in claim **26**, wherein the value of N at said reference magnitude or degree of mapping resolution, corresponds to one of the following values: 512°, 256°, 128°, 64°, 32°, 16°, 8°, 4°, 2°, 1°, 30′, 15′, 7.5′, 3.75′, 1.875′, 0.9375′ and 0.46875′.

28. A method for providing an electronic map generating system for generating reproductions of a map with selectable degrees of mapping resolution, said method comprising the steps of:

storing a plurality of computer files containing mapping data corresponding to respective surface areas of a mapping surface, wherein said plurality of computer files is organized into a plurality of successive magnitudes, each magnitude for presentation of said mapping data with a different degree of mapping resolution from a first or highest magnitude with lowest resolution to a last or lowest magnitude with highest resolution, files of a respective magnitude including mapping data which are appropriate to a degree of mapping resolution being afforded at said respective magnitude while excluding mapping data which are not appropriate to said degree of mapping resolution, and wherein a predetermined file naming procedure is utilized to assign, to each respective computer file, a unique filename which:

relates said respective computer file to all other computer files having mapping data corresponding to a same magnitude or degree of mapping resolution; and

relates said respective computer file to any computer file comprising mapping data corresponding to a same surface area of a mapping surface as said respective computer file; and

accessing said plurality of computer files using said predetermined file naming procedure, to generate a reproduction of a selected area of a map at a selected degree of mapping resolution.

29. A method as claimed in claim **28**,

wherein each said unique filename is represented by a value contained in a plurality of bits, and

wherein said predetermined file naming procedure;

utilizes a first predetermined subset of said plurality of bits to relate said respective computer file to all other computer files having mapping data corresponding to a same magnitude or degree of mapping resolution; and

utilizes a second predetermined subset of said plurality of bits to relate said respective computer file to any computer file comprising mapping data corresponding to a same surface area of a mapping surface as said respective computer file.

30. A method as claimed in claim 29, wherein said unique filename also includes geographical information which can be used to relate a geographical coordinate position of a respective computer file with respect to a coordinate layout of surface areas of said mapping surface.

31. A method as claimed in claim 28,
wherein an assignment of said unique filenames using said predetermined file naming procedure results in said respective computer files of said plurality to be related in a quadtree database structure.

32. A method as claimed in claim 31, wherein the respective area of a mapping surface covered within the computer files of consecutive magnitudes or degrees of mapping resolution changes at a predetermined rate in that, when a computer file at a reference magnitude or degree of mapping resolution contains mapping data corresponding to an N×N area of a mapping surface (where N is a real number, and is associated with one of the conventional degree °, minute ′, or second ″ mapping scale divisions), then a computer file at a next consecutive magnitude having a higher degree of mapping resolution contains mapping data corresponding to an (N/2)×(N/2) area of said mapping surface.

33. A method as claimed in claim 32, wherein the value of N at said reference magnitude or degree of mapping resolution, corresponds to one of the following values: 512°, 256°, 128°, 64°, 32°, 16°, 8°, 4°, 2°, 1°, 30′, 15′, 7.5′, 3.75′, 1.875′, 0.9375′ and 0.46875′.

* * * * *

 Unified Patents Exhibit 1005 App'x A-N