Responsive Thinwire Visualization of Large Geographic Datasets

by

Kenneth Been

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Department of Computer Science
New York University
September 2002

Approved: _____
Chee Yap

Acknowledgments

A special thank you goes to my advisor, Chee Yap, who has been a wonderful guide and collaborator. I hope to continue working with him, on this and other projects.

Thanks also to the other members of my thesis committee, Richard Cole, Dennis Shasha, Arthur Goldberg and Denis Zorin. Their comments and critiques pushed the dissertation toward its final form.

Thanks to Alan Siegel and Richard Cole for inviting me to work with them on legislative redistricting, which was my first exposure to TIGER data.

Thanks to Zilin Du, Yan Koyfman and Josh Harmon, who contributed code and insight to the project.

Thanks to Kia Makki and Niki Pissinou, who convinced me to go for a Ph.D. in the first place.

And finally, thank you to my family and near-family, especially Ruth and Irwin Been and Mary Lynn Miller, for all their loving support and wisdom.

Abstract

This thesis describes a web-based, responsive, zooming and panning visualization system for a full-featured geographic description of the United States. Current web-based map servers provide, from a visualization standpoint, little more than one static image per page, with hyperlinks for navigation; continuous zooming and panning requires locally stored data. Our primary contribution is a multi-threaded, scalable and responsive client-server architecture that responds to user requests as naturally and quickly as possible, regardless of network bandwidth reliability. This architecture can be generalized for use in other applications, including non-geographic ones. To this we add a scalable and flexible user interface for navigation of multi-scale geographic data, with intuitive zooming and panning, pop-up feature labels, and a user controlled tree-hierarchy of windows. We build software tools and algorithms for translating the U.S. Census Bureau's TIGER data into a format designed for speedy database retrieval and network delivery, and for generalizing the data into multiple levels of detail. Because of anomalies in the TIGER data, this processing requires some human intervention.

Contents

Acknowledgments **iii**

Abstract **v**

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**
1.1 Background and Motivation 1
1.2 Thesis Overview 8
1.3 Terminology 9

2 Runtime System Architecture **13**
2.1 Introduction and Prior Work 13
2.1.1 Hypertext Transfer Protocol 13
2.1.2 Walkthrough Applications 15
2.1.3 Remote Databases 16
2.1.4 Foveated Image Visualization 17
2.1.5 The Petra-Flow Framework 18
2.2 Design Constraints 18
2.2.1 Constraints Related to Problem Definition 19
2.2.2 Simplifying Constraints 19
2.2.3 Atomicity Constraints 20
2.2.4 Java Related Constraint 21
2.3 Thread Architecture 21
2.3.1 Data Flow 23
2.3.2 Command Flow 25
2.3.3 Note on the Event Handler Thread 28

2.3.4 Thread Behaviors 29
2.3.5 Discussion 33
2.4 The Priority Queues 34
2.4.1 The Basic Priority Queue 36
2.4.2 The Renderer Priority Queue 38
2.4.3 The Response Sender Priority Queue 39
2.4.4 Data Provider Priority Queue 43
2.5 Responsiveness 43
2.5.1 Theoretical Arguments for System Responsiveness 44
2.5.2 Experimental Testing of System Responsiveness 46
2.6 Cache Consistency 50
2.6.1 Some Flawed Approaches 50
2.6.2 A Correct Approach 53
2.6.3 Proof of Correctness 53
2.7 Discard Policy 56
2.8 Searching the Cache 58
2.8.1 Requirements 59
2.8.2 The R-tree 60
2.8.3 The QR-tree 61
2.8.4 The QR-tree and Concurrent Access 64
2.8.5 Tree of Trees 66

3 Preprocessing the Dataset **69**
3.1 Related Work 69
3.2 The TIGER Data 71
3.2.1 Overview 71
3.2.2 File Organization 72
3.2.3 Anomalies 74
3.3 Merging the Raw Data 82
3.4 Database Schema 84
3.4.1 Goals 84
3.4.2 The Schema 85
3.4.3 The Intermediate Database 86
3.5 Processing the TIGER Files 89
3.5.1 Goals and Obstacles 89
3.5.2 Deducing the Polygon Containment Hierarchy 90
3.5.3 Processing the County Borders 93
3.5.4 Temporary Data 94

3.5.5 Protocol for Producing S-Level 0 . . . 95
3.6 Generalization and Building the LOD Hierarchy . . . 96
3.6.1 Manual Generalization . . . 97
3.6.2 Automatic Generalization . . . 98
3.6.3 Grouping the Counties . . . 99
3.6.4 Protocol for Producing S-Level i, i > 0 . . . 102
3.7 Producing the Blobs . . . 103

4 Visualization Interface **105**
4.1 Related Work . . . 105
4.1.1 Multi-Scale Navigation . . . 105
4.1.2 Labeling . . . 106
4.2 Our Windowing User Interface . . . 108
4.2.1 Anatomy of a Window . . . 108
4.2.2 Relationships Between Windows . . . 113
4.2.3 Global Options . . . 114

5 Conclusions and Future Work **117**
5.1 Performance Improvements . . . 118
5.2 Visualization Interface . . . 120
5.3 Other Versions . . . 121
5.3.1 HTTP Tunneling . . . 121
5.3.2 Cell Phones and PDAs . . . 122
5.3.3 Responsive Client-Server Kernel . . . 123
5.4 Dataset . . . 123

Appendix **125**

Bibliography **128**

List of Figures

2.1 Runtime System Data Flow Diagram 23
2.2 Runtime System Command Flow Diagram 25
2.3 The Basic Priority Queue 36
2.4 A Heap on the Renderer Priority Queue 40
2.5 The Response Sender Priority Queue 41
2.6 Polygon rendering time vs. number of points in polygon and descendants 49
2.7 Sample LOD to Window Relationship 57
2.8 Sample Preserve Ranges 58
2.9 QR-tree Node with Children 61

3.1 TIGER Data Diagram 73
3.2 Statue of Liberty National Monument 75
3.3 Prince William County, Virginia. Contains a hole, islands inside the hole, and a strip dividing the hole. 77
3.4 Manassas Park, Virginia. Fits inside the upper right portion of the hole in Prince William, straddling the strip across that hole. . 78
3.5 Manassas, Virginia. Fits inside the hole in Prince William, and the hole in the middle of Manassas can just barely be seen. 79
3.6 Closeup of the islands in the middle of Prince William County, Virginia. 80
3.7 Closeup of Manassas, Virginia. The hole in the middle, and the two outlines next ot it, match the two islands in Prince William County. 81
3.8 Before and After Merging 83
3.9 Cluster and Family Diagram 91
3.10 Maryland, Area Acquiring Algorithms 1 (left) and 2 100
3.11 Maryland, Area Acquiring Algorithms 3 (left) and 4 100
3.12 Maryland, Area Acquiring Algorithms 5 (left) and 6 101

3.13 Line Simplification Anomalies: (a) child polygon outside its parent, (b) line crossing itself. 103

4.1 User Session 1 109
4.2 User Session 2 110
4.3 User Session 3 111
4.4 Client with Place Editor 116

5.1 Client-Server Kernel 123

List of Tables

2.1 Times in seconds between clicking the first jump button and system coming to rest, as measured by human with a stopwatch. . . 48

3.1 Blob Database Statistics 86
3.2 Numbers of geographic objects in intermediate database, after merging, at each S-Level 88
3.3 Database Sizes, in Mb 88
3.4 Generalization Characteristics of Each S-Level 98

Chapter 1

Introduction

1.1 Background and Motivation

In the mid 1980s Silicon Graphics Inc. introduced workstations with sophisticated graphics hardware for between $40,000 and $100,000; in the late 1990s advanced graphics capabilities became available on most common PCs. Card, Mackinlay and Shneiderman [9] identified these two moments as the beginning and ending points of the "foundational period" for visualization research. "The path is now clear", they say, "for information visualization to be used in mass market products." ([9] p. xiii) The original motivation for the current research was to bring the benefits of quality visualization to a particular "mass market" activity: browsing road maps on the Internet.

In [9], *visualization* is defined as "the use of computer-supported, interactive, visual representations of data to amplify cognition", where *cognition* is defined as "the acquisition or use of knowledge". Visualization is roughly divided into *scientific visualization* and *information visualization*. Scientific visualization [56] is concerned with data that is based in physical reality. Examples include the visualization of fluid flow through a pipe, or the structure of a molecule. Such data usually has a natural geometric representation. Information visualization, on the other hand, is concerned with abstract data that is not physically based and not inherently geometrical, such as financial data or collections of documents. In this case, there is the challenge of designing a clear and meaningful *spatial* representation for the non-spatial abstractions; a problem that does not arise in scientific visualization. The geographic data that is the concern of this thesis falls largely into the scientific visualization category, since the objects are mapped from the surface of the earth, but it also contains abstract semantic data, such as feature

names ("Hudson River") and feature types ("Primary Highway With Limited Access") that is not inherently geometrical, giving the problem some aspects of information visualization. We are primarily concerned with those aspects of visualization that are common to both scientific and information visualization: producing quality interactive visual representations of data. The quote given above, about information visualization being used in mass market products, certainly applies just as well to scientific and geographic visualization.

The most obscure part of the visualization definition above is probably the phrase "amplify cognition". Interestingly, visualization is sometimes referred to as "externalization" (in [90], for example), the idea being that a visual representation of some piece of knowledge is placed in the external world (external to the mind), where it can be used more easily. Consider the problem of multiplying two numbers of two or more digits each. Most of us find it easiest to perform the multiplication using a pencil and paper, rather than doing it in our heads. The reason is not that the task involves any particularly difficult sub-tasks—in one common method learned by many schoolchildren the only sub-tasks required are the multiplication of two single-digit numbers and the addition of two single-digit numbers. The problem, rather, is that our short-term memories are not good at saving the partial results until such time as they are needed. The pencil and paper augment our cognitive abilities specifically by increasing the capacity of our short-term memories. Notice that the pencil and paper example fits our definition of visualization in every respect except that it is not "computer-supported". Clearly, computers provide us with many more opportunities to amplify cognition.

There has been much interest in recent years in geographic visualization, or *geovisualization*, and even virtual reality interfaces to geographic data. This includes the ViRGIS system of Pajarola, et al. [65, 67, 64, 63, 87], and the GeoVISTA project directed by MacEachren [52, 85, 53, 86]. These projects strive to let the user smoothly navigate around a visual representation of a given geographic region. The smooth navigation is in service of the ultimate goal, which is still to amplify cognition: MacEachren says that geovisualization "is about the use of visual geospatial displays to explore data and through that exploration to generate hypotheses, develop problem solutions, and construct knowledge" [52]. (By "explore" he does not mean exclusively navigation, but it is certainly one of the methods of exploration. Other explorations might include interactive queries of a database for population data, etc.) A related problem to geovisualization is the one of architectural walkthroughs; see the work of Funkhouser, et al. [32, 31, 29, 33, 30]. The primary differences between geovisualization and

walkthrough are that (1) the walkthrough data is 3D, as opposed to the 2D or 2.5D geographic data, and (2) the walkthrough data is usually assumed to be purely geometric. The existence of abstract, non-geometric aspects of geographic data explain the paradoxical situation where the 2D problem is harder than the equivalent 3D problem. Nonetheless, the visualization goals in geovisualization and walkthroughs are similar in many ways.

Consider, for a moment, implementation. There is an assumption, whether explicit or implicit, in the geovisualization and walkthrough literature, that the visualization is composed of a sequence of *frames*—a model taken from film and video. For performance, then, they seek to guarantee a minimum *frame rate*. Funkhouser [29] explicitly targets a frame rate of ten frames per second. What this means is that there is another assumption, whether explicit or implicit, of a minimum bandwidth separating the data from the rendering engine. When data is stored locally, this would normally be the transfer rate between secondary memory and main memory. But even when the data must be retrieved from across a network, such as in the work on remote walkthroughs by Schmalstieg [78, 77, 79, 43], Teler and Lischinski [88], and Zach and Karner [107], a minimum bandwidth is assumed, though it is allowed to be much lower than in walkthrough of locally stored data.

In this thesis we are concerned with the visualization of data that is stored across a *thinwire*, a term that we use to describe bandwidth that is not only low but also unpredictable—it might drop to zero temporarily, if the network becomes disconnected. Forman [26, 27] describes such a bandwidth as "variable and volatile". In a thinwire setting, we cannot assume a minimum bandwidth. Normally we think of the thinwire as the Internet, and in today's world, in which many people connect to the Internet with cell phones and PDAs over wireless networks, network connections are as volatile as ever [28]. Techniques developed for walkthroughs, such as multiresolution modeling [41], progressive transmission [44, 79, 66, 72], prefetching [36, 46, 47, 19, 15, 68], caching [68, 15], visibility culling [89, 107], and others [18, 54], are not sufficient in a thinwire environment, though they may still be useful. In this environment, we need to consider *responsiveness*; but before discussing responsiveness in detail, we first consider some general issues of interactivity and visualization.

Card, et al. [9], discuss the question, "How fast does interaction have to be?" They delineate three levels of interaction: the 0.1 second level, the 1 second level, and the 10 second level. Two events that occur within 0.1 second of each other "fuse into a single percept". This is the speed that is necessary for the perception of cause and effect relationships—if a button on screen changes state within 0.1

second after being clicked, then we perceive that the click caused the change of state. (Note that *perceiving* a relationship is more immediate than *guessing* or *deducing* one.) The illusion of animation breaks down if each frame lasts more than 0.1 second—recall Funkhouser's target frame rate of ten frames per second. The 1 second level is the pace of human dialogue, the time for an *unprepared response*.

> If a speaker pauses for more than about a second, the listener feels compelled to say "uh-huh" or nod to assure the speaker the listener's end of the communication channel is intact. If the speaker cannot think of what to say for more than about a second, the speaker will say "ah" or otherwise indicate that his or her end of the channel is still working. [9]

The 10 second level, which is really a range of 5–30 seconds, represents the time needed for a "minimal unit of cognitive work". Examples include selecting and modifying a small section of text on the screen, or making a routine move in chess (not one that requires lengthy study of the board).

This has implications for our geographic visualization system. Since we are in a thinwire environment, we do not have the option of giving the user all the data they need at a rate that is high enough for a useful animation or video-like display. However, the primary thesis of this dissertation is that we can still build a system that will provide the user with a high quality visualization experience. What must be the nature of such a system? First, if it is possible that a response will take more than a second, there should be some indicator on the display that lets the user know that the system is working on the response, and will be answering it shortly. This is equivalent to a speaker saying "ah" when having to pause for more than a second to think about what to say. Second, consider another human dialogue analogy. Suppose A asks B a question, but before B finishes answering, A interrupts with a different, more urgent question. Assuming A asks the second question politely, B will normally go ahead and answer the second question first, and then maybe return to answering the first question, if the information is still relevant. Similarly, our visualization system, which assumes that all users are polite, should always give highest priority to answering the most recent requests. Third, whenever the user makes an input action with the mouse, we must ensure that the action is recognized and acknowledged as soon as possible. The user should ideally perceive the acknowledgment within 0.1 seconds of making the action, and this should be the case no matter how busy the system is with handling previous input actions. (In earlier versions of our

system we found that even delays in acknowledging mouse drags of as little as half second or so were quite disorienting.) These last two issues lead us to the question of responsiveness.

As Forman [27] points out, "scaling applications to fit resource constraints is an old problem". If an application is scaled to a fast system, by providing capabilities that a fast system can handle, it will not have good response times on slower systems; if it is scaled to a slower system, providing limited capabilities, it will not be used to its full potential on faster systems. "Traditionally," he says, "programmers have managed this balance, often unawares, by *static sizing*: they scale the amount of processing their application requires based on its response time on a specific platform, typically their own."

The obvious incremental improvement to static sizing is *dynamic sizing*, in which applications are written to be able to adjust their service levels to match the current environment [27, 26]. This is the idea behind "quality of service" negotiation, in which a client and server undergo a negotiation protocol to decide what level of service can be supported [14, 71]. The problem with this, as Forman notes [27], is that it requires

> (1) applications capable of multiple service levels, (2) a predictive model of the response time/workload at each service level, (3) a mechanism to sense the current resources of the environment for selecting a service level, (4) optionally, the capability to make resource reservations, which may have to be revoked later, (5) the capability to monitor an ongoing operation to detect early that it will not meet its objective due to either changes in the environmental resources/reservations or errors in the performance model, and (6) a mechanism to cancel and renegotiate an operation at a different service level, preferably not throwing away work that has already been accomplished.

Furthermore, dynamic sizing breaks down when bandwidth is highly volatile, because the bandwidth measurement might already be out of date by the time the service request is made. What we really need are systems that function in volatile and unpredictable resource environments, such as thinwire connections. We characterize such systems as *responsive*.

In this thesis, responsiveness refers to an intrinsic property of an application itself, not of the application's performance under a particular condition (see also [105]). An application that gives good response times under high bandwidth conditions would only be considered responsive if it had the capability built in to also give good response times during low bandwidth times. Of course, if

bandwidth temporarily drops to zero we do not expect the system to generate new data, but it should at least not freeze, and should continue to provide whatever useful responses that it can. In that case the responses themselves would not be as good, but the point is that the system should "feel" responsive, even when not every request can immediately be satisfied. Responsiveness implies the ability to be flexible and accommodate the user's whims and mind changes, so that the most recent request should take priority over older unsatisfied requests. Forman [26, 27] identified four principal techniques, or primitive elements, for providing responsiveness in a volatile resource environment:

Incremental results and feedback before completion Users make new requests based on partial responses to old requests.

Concurrency Multiple tasks operate asynchronously, so that one slow task does not hold up all the rest.

Dynamic prioritization Outstanding tasks can have their priorities changed based on more recent user actions.

Cancellation Outstanding tasks may be canceled if they are made unnecessary or irrelevant by more recent user actions.

Notice that three of these primitives can be directly mapped into our human dialogue analogy: feedback before completion corresponds to a listener interrupting with a new question before a previous answer is finished, dynamic prioritization corresponds to a speaker accepting the (polite) interruption and answering the most recent question first, and cancellation corresponds to the speaker dropping the response to an old question if newer questions have made it no longer relevant. The system that will be described in chapter 2 makes use of each of these techniques in a way that is specially tailored to thinwire geographic visualization.

One last issue that must be faced by any large scale geographic visualization project is the largeness of datasets. Since secondary storage devices are now available that are larger than most geographic datasets, the physical size of the dataset is not *itself* an issue, but it is indirectly an issue because (1) we must devise storage and indexing schemes that allow queries to be performed quickly, and (2) we must provide a visualization interface that allows a user to navigate data at a very large range of scales—in other words, be able to see details, without losing context.

In their system for spatial data management on the Netherlands Cadastre, van Oosterom and Lemmen [99] are working with a database of 50 GB (and

growing). They use techniques and data structures developed in [93, 102, 100, 76, 94, 96, 97, 95, 98] for storing and manipulating data at a very large range of scales in a single dataset, using an object-relational DBMS. However, that system does not include the zooming/panning visualization interface that we seek. Our dataset is more modest, but our visualization goals have higher performance requirements. The raw data that we download from the US Census Bureau requires 3.8 GB of disk space, although after preprocessing it only requires 1.2 GB in the database. In both of those formats the data is compressed. While this may not be large in an absolute sense, it is large for a thinwire geographic visualization system. By comparison, the system described in [85] saw significant performance deterioration when the United States was viewed at only the county level; the walkthrough system of Shou, et al. [81], for a "very large virtual environment", required only 200 MB of disk storage.

Equally important, our data is *conceptually* large: from an overview of the entire contiguous continental United States (48 states plus the District of Columbia), down to street level detail. In discussing the overview + detail interface, in which one window shows an overview of the data while a second shows details of one small portion of it, Card, et al. [9], point out that "typical zoom factors in papers on overview + detail strategies are 5–15", but intermediate views can be introduced "to support zoom factors of 100 or 1000". Our data set requires a zoom factor of almost 10,000 from one extreme to the other.

To handle this large zoom factor, we need a *level of detail*, or *LOD*, hierarchy, so that the amount of details shown can be adjusted to the current zoom level. When we are zoomed out the screen would be much too crowded if all of the details were shown. This introduces the problem of map *generalization*, which is the creation of a coarse map out of a detailed map. There are several primitive methods used in generalization; the two that we use are object deletion, in which small or unimportant objects are removed from the map, and line simplification, in which the number of points used to describe a polygonal line is reduced. Other methods that we do not use include replacing a graphical representation with a symbol, and replacing a polygon with a line or point, as might be done with a river or pond, respectively. In cartography, generalization tends to be a highly subjective operation [104], which distinguishes it from the walkthrough domain, where the purely geometric data can often be generalized automatically.

1.2 Thesis Overview

The contributions of this work are as follows. We have designed a client-server system that is highly responsive in a thinwire setting; the primary components of this system as relates to responsiveness are a multi-threaded architecture and a sophisticated form of priority queue. We have developed data processing techniques that allow us to transform raw geographic data into an optimized format that allows fast query performance at runtime. We have designed a flexible, scalable user interface that allows a user to navigate through an extremely large data space—one having a maximum zoom factor of almost 10,000—without losing context or "getting lost". We have combined all of these techniques together to successfully build a responsive thinwire visualization system for the contiguous continental United States (48 states plus the District of Columbia). As far as we know, this is the first web-based visualization system to provide smooth zooming and panning for anywhere near such a large dataset and high zoom factor, and the first web based visualization of any size and any kind to incorporate such a high degree of responsiveness. As of this writing, a demo is available on the web [61].

Chapter 2 covers the techniques used in the runtime system to make the system responsive—a multi-threaded client-server architecture, a sophisticated form of priority queue, and others. We argue that under some reasonable simplifying assumptions the system is in some sense maximally responsive. We also prove the correctness of our data exchange and cache consistency protocols, which is not obvious because of the reordering and canceling of tasks.

Chapter 3 describes how the data is preprocessed into a format that can meet the performance demands of the runtime system. This is a more difficult problem than it may seem, owing primarily to the many irregularities that are found in the raw data. Such difficulty is not unusual; similar problems were encountered by Funkhouser [33] and Kernighan and Van Wyk [45] in processing architectural data. In addition to translating the data, chapter 3 also covers generalization. We have experimented with doing manual generalization, to produce high quality maps, but because of the high amount of labor needed for manual generalization, we have instead developed a largely automatic process.

Chapter 4 describes our visualization interface, which provides intuitive and convenient tools both for data exploration and keeping relevant information visible as context for the exploration. We employ a novel system of multiple linked windows, which gives the users great flexibility for navigation, and is also scalable enough to easily handle our large zoom factors.

1.3 Terminology

For convenient access, we list here, in alphabetical order, a number of terms that are used in various parts of the thesis, and give brief descriptions.

blob A list of objects, that have compressed into a byte array. The name comes from the database notion of *binary large object*, although in the database the blobs are stored as simple byte arrays—we do not use PostgreSQL's large object interface. The blobs are created in the preprocessing phase, and never changed.

blob id Each blob is given a unique id number during the preprocessing step. These are used by the server to avoid sending duplicate information to the client.

cache The temporary store of geographic objects in memory on the client side.

cache copy This isn't really a copy; it is a temporary store, in memory on the server, of the blob ids of the blobs whose contents are currently in the cache.

child polygon In the *polygon containment hierarchy*, a polygon that is contained within another *parent polygon*.

data/commands Later, in the description of how information and flows through the system, we will distinguish between the flow of data and the flow of commands. This is done mainly for clarity of explanation, and the distinction is somewhat arbitrary. Both data and commands are encapsulated in tasks. As "data", we count tasks that are directly involved in passing the content (what the user wants to see) from the server side database to the client side windows. This includes objects, blobs, blob ids, packages, ranges (as requests), and mouse events. As "command", we count tasks that are for a higher level control of system behavior; for example, the add-window command is passed through the system to inform all threads that the user has added a new window. The full list of commands will be given in section 2.2.

database The permanent store of geographic objects on the server side.

LOD Level of detail. This is a somewhat general term; we also define and use the more specific terms *pLOD* and *S-Level*.

MBR Minimum bounding rectangle.

merging In the preprocessing phase, combining two or more objects into one. See section 3.3.

object A geographic feature: polygon, line or landmark.

package Like a blob, a package is a list of objects. Packages differ from blobs in that the objects are not compressed; the objects are stored in a queue rather than an array; and packages are created at runtime on the client side, out of objects that are pulled out from the cache because their ranges intersect a range request.

parent polygon In the *polygon containment hierarchy*, a polygon that has another *child polygon* completely inside it (i.e., the parent polygon has a "hole").

pLOD Pixel level of detail. This is a kind of scale, mapping real world coordinates to screen coordinates, in meters per pixel.

polygon containment hierarchy In the data, some polygons are contained within others, forming a hierarchy of parent/child relationships—a forest. This information needs to be available when the polygons are rendered to the screen, so that they will be rendered in the proper order (parent first).

preserve range When the cache and cache copy are *pruned*, all elements that do not intersect with the preserve range are removed.

priority queue We create several instances of a sophisticated form of priority queue to perform two functions: (1) coordinate inter-thread communication, and (2) order the tasks according to priority. Several of the threads get their input tasks from priority queues. Of those that don't, two get their tasks from the network (or thinwire), and one, the GUI event handler thread, gets its input tasks from a simple queue. The event handler and its input queue are part of the Java runtime environment, so we do not have the option to replace them with a different queuing approach. When there is no possibility of confusion, we sometimes refer to a priority queue as simply "queue".

prune To keep the cache from getting too big, the client will occasionally initiate the *prune* process, by which items that are deemed to be superfluous are removed from the cache and the cache copy.

range A rectangular range of latitude/longitude coordinates. Ranges are used for specifying requests for data, and for describing the "location" of a geometric feature. Also sometimes referred to as *bounding box* or *minimum bounding rectangle* (MBR).

request A request is a pair, $(range, level)$, that is sent to the server, indicating that the client wants to receive all objects that intersect the given range, at the given level of detail. Also sometimes referred to as *range request*.

response The response that the server returns for a range request may include zero or more blobs, and should not include any blob that the client already has received (and not removed from its cache).

query The query that the server sends to the database in response to a range request. Takes the form "select all blobs from the given level of detail whose ranges intersect the query range and whose blob ids are not in a given set".

S-Level Simplification level. A range of *pLODs* to which we associate a single set of geographic objects in the database. When users zoom from one pLOD to another within the same S-Level, they see the same objects, but at a different magnification; when they zoom to a pLOD in a different S-Level, they will see a different set of objects.

task A task is a record that contains information that functions as an instruction to a thread to perform some action. The task may also contain data. Threads communicate with each other by passing tasks. We also sometimes use the term "task" to refer to the action that a thread does, as per the instructional information in a task record.

thinwire Normally this is the Internet; more generally, it can be any communication link between the client and server portions of the system, and is assumed to have unreliable bandwidth. Messages sent from one entity to another across the thinwire are assumed to arrive in first-in first-out order.

thread The system uses several long-lived, lightweight threads on both the client and server; these are the focal points of system behavior. The client side threads are created when the client is started, and live until the client finishes. When a client requests a connection with the server, a master thread on the server side starts three new threads that are dedicated to this client. These threads live until the client disconnects.

top-level polygon A polygon that has no parent in the *polygon containment hierarchy*

window The user views the geographic data through a number of windows. The windows are organized as a tree; the relationship between parent and child windows is described in chapter 4.

Chapter 2

Runtime System Architecture

In broad outline, we want a system for visualizing a large geographic dataset that does not reside on the user's computer. The user should have the capability to open multiple windows for different viewpoints on the underlying data. The system must have zooming and panning capabilities, and has responsiveness as a primary goal. This is a very open-ended specification, seemingly allowing an unlimited number of possible solutions. We start this chapter by looking at how others have tackled similar problems. Then, in sections 2.2, 2.3, and 2.4 we'll describe in detail the architecture that we have designed. Section 2.5 contains a discussion of the responsiveness that is built in to the architecture. And the last three sections cover the client and server side caches: how they work, and why they remain consistent.

2.1 Introduction and Prior Work

We are interested in the general class of interactive client-server situations wherein a user at the client end makes multiple requests and a server replies with responses of some form. There may be zero or more responses to each request. The goal is always responsiveness. We do not consider streaming media, because that is a non-interactive paradigm.

2.1.1 Hypertext Transfer Protocol

HTTP, the Hypertext Transfer Protocol, is the nearly ubiquitous protocol used for common web browsing. Under this protocol, a client requests web resources (HTML pages, images, etc.) and the server returns the requested resources.

HTTP/1.0 [6] encapsulates the HTTP functionality generally in use on the web going back to 1990, though it was never an official standard. In this version of the protocol, each request/response transaction requires its own TCP connection. There is no limit on how many concurrent connections a browser might have open to a server, and the order in which responses to concurrent connections arrive is not specified. This approach made some sense in the early days of the web when most pages were simple HTML text, but as the web matured pages became filled with multiple images and other resources, and the overhead of starting a new connection for each resource request became a serious problem for both perceived responsiveness and web traffic [84].

As an ad hoc fix for some of these problems, many HTTP implementations allow persistent (or "keep-alive") connections, which are not closed after a request/response transaction, so that multiple transactions can take place in sequence without opening a new connection. The connection opening overhead is reduced, but parallelism between connections can also be lost, which raises the issue of how many connections to open. The Mozilla Pipelining FAQ [62] briefly discusses this and other issues, although in the context of HTTP/1.1.

To answer some of the above problems, HTTP/1.1 was proposed as a formal standard [24]. For our purposes, the relevant changes in HTTP/1.1 are that persistent connections are now the default behavior, and "pipelining" of requests and responses on a connection is allowed. With pipelining, multiple requests can be sent on a connection without waiting for each response. The responses must be returned in the same order that the requests were received in. Clients are requested to not open more than two simultaneous connections to the same server. The overhead and inefficiency of opening, starting up, and closing TCP connections is reduced, as compared to HTTP/1.0, but without as much parallelism of multiple connections open simultaneously. Nielsen, et al. [60] ran some tests on HTTP/1.0, HTTP/1.1 without pipelining, and HTTP/1.1 with pipelining. The number of simultaneous open connections was limited to six with HTTP/1.0 and one with HTTP/1.1. They found that HTTP/1.1, with or without pipelining, reduced the number of bytes and packets transmitted, but for response latency, HTTP/1.0 actually outperformed HTTP/1.1 without pipelining. HTTP/1.1 with pipelining gave the best performance for all three measures.

Some consideration was given to a more advanced protocol, called "HTTP-NG" (for "next generation") [59]. An early draft proposal for HTTP-NG contained the option for a client to cancel a request [83], but this does not seem to have made it into the later drafts. In any case, for various reasons, including a feeling that it was "too early" and people "were just getting used to HTTP/1.1",

work on HTTP-NG is no longer being carried out [103]. It appears that canceling and reordering requests will not be part of HTTP for the foreseeable future.

2.1.2 Walkthrough Applications

Architectural walkthrough applications tend to be governed by a video-style "frame rate" paradigm. In the approach of Funkhouser, et al. [32, 29, 33, 30], functional operations are grouped into pipeline stages, with each stage implemented by a thread. The flexible implementation allows the grouping of operations into stages to be specified by the user, so different groupings can easily be tested. The functional operations are User Interface (U), Visibility Determination (V), Detail Elision (D), Rendering Operations (R), Lookahead Determination and Cache Management (M), and Database Input/Output Operations (I). The order of operations is predetermined, but the grouping can be specified. The pipeline begins with U and ends with R, and in between is split into two forks, VD and MI. So example groupings of the operations on the "lower" fork are (UV)(DR) and (UVD)(R). Each thread may run on a separate processor, but the data is assumed to be stored locally, so there are no client-server considerations. The threads communicate with each other using simple queues; there is no attempt to cancel or reorder requests, and it is assumed that all requests are fully answered. The goal, then, is to make the pipeline fast enough so that a given frame rate can be maintained. This approach can clearly not be used when we have thinwire considerations.

Schmalstieg and Gervautz considered the problem of walkthrough with remote data (i.e., over a thinwire) [78, 77]. They use a system architecture that is superficially somewhat similar to ours. The client has a network manager, a database manager, and a manager for rendering and the user interface; the server has a connection manager and a database manager. User requests are captured by the UI manager, passed to the client-side database manager which decides if the request needs to be passed to the server (if the data is not in the local cache). Requests that must go to the server are passed to the network manager, across the network to the connection manager, and finally to the database manager, which finds the data and sends it back along the same path to the client rendering manager. Apparently there is no option to cancel or reorder requests, although the connection manager on the server side contains a priority queue, whose use is not explained.

2.1.3 Remote Databases

In the ViRGIS (Virtual Reality GIS) and ViRXIS (Virtual Reality based Information Systems) projects [65, 67, 63, 87], which aim for a virtual reality interface to a GIS or other information system, the client-server communication appears to consist simply of queries on a remote database. The server does contain a thread (or process) apart from the DBMS for receiving the requests and converting them into queries, but there is no mention of it doing any "smart" managing of the requests; it simply passes them to the DBMS. There is in general no special consideration given to the slowness of the network, though they say that the system could run on a wide-area network. (Their implementation was on a local-area network.)

Two other projects designed architectures for efficient prefetching from remote databases. In the system of Gerlhof and Kemper [36] the server has two threads and two sockets for each client: one socket/thread for demand requests, and one socket/thread for prefetch requests. The two types of requests are given equal priority. The client has two sockets, a demand socket and a prefetch socket, for connecting to the analogous socket at the server, and three threads: Application, Prepager, Receiver. The Prepager writes prefetch requests to the prefetch socket, and the Receiver reads the prefetched pages on the same socket. The Application thread both writes requests and reads responses on the demand socket. Prefetch requests can be pipelined, and the server must return the prefetch pages in the same order that they are requested. Requests cannot be canceled. The Application thread blocks if the page it wants to access is not available yet. In this situation, there are two possibilities: either a prefetch request has already been sent for that page, or not. In the first case the Application thread simply blocks, and when that page comes back to the Receiver, the Receiver will unblock the Application thread and cede control to it. In the second case the Application thread sends a demand request, and waits for the response. While the application thread is blocked, the system cannot respond to the user, so system responsiveness is dependent on the quality of the prefetching technique.

In Knafla's system for prefetching from a remote database [46, 47], the client has an App Thread, a Prefetch Thread, zero or more Support Threads, and a Flush Thread. Each thread has one associated socket for connecting to the server, presumably by a standard database connection. (It's not clear if each Support Thread has its own socket, or if there is one socket for all the Support Threads to share.) In addition to its buffer pool, the client keeps a prefetch list, which identifies pages that have been prefetch requested but are not in the buffer pool yet (i.e., have not been received yet). The App Thread is the main

application thread, and has the highest priority. When it needs a page that is not in the buffer pool or the prefetch list, it sends a demand request to the server, and waits for the response. The Prefetch Thread has medium priority. It checks which objects the app thread is processing and determines which pages should be prefetched. If more than one page should be prefetched simultaneously, it starts one or more Support Threads, and each Support Thread requests one prefetch page from the server. Support Threads have lowest priority. The Flush thread flushes dirty pages to server. The server is not multi-threaded; it performs each request sequentially. Multi-threading the server "would further improve the whole systems performance and is part of future work". As with the Gerlhof and Kemper approach, the App Thread blocks when it does not have a page it needs, so system responsiveness is dependent on the quality of the prefetching technique.

2.1.4 Foveated Image Visualization

Chang, Yap and Yen [12] designed a client-server system for visualization of very large "multi-foveated" images using wavelet techniques. The client requests various pieces, at various resolutions, of an underlying image that is too large to view at full resolution all at once (and too large to fit into main memory). The server has only one thread, and simply responds to every request in the order that it is received. The client has three threads: network, display and manager. The network thread handles all communication with the server: sending requests and receiving responses. The display thread captures user input and handles display output. The manager thread is the "brain" of the program; among other things, it translates user input into requests for the network thread (and hence the server), and notifies the display thread when new data comes in from the server. There is always at least lowest resolution data available for any part of the underlying image, so the display thread can always supply some kind of response to the user, no matter how slow the server or network is, a crucial property for responsiveness. However, there is no capability for canceling or reordering requests; every request is handled in the order that it is received, even though this might not be the user preference.

Yap and Yen [106] later improved on the above architecture. In the improved client, the network thread is split into two threads, one for reading and one for writing; the display thread is split into two threads, one for user input and one for display output; and the manager thread is eliminated, its responsibilities divided among the four new threads. The server was not changed. This version allows

pipelining of requests, both at the network and at the user input stage. In other words, multiple requests can be sent to the server without waiting for each response, similar to pipelining under HTTP. And similar to HTTP, the server must still respond to each request in order. At the user input stage, multiple mouse motions might be captured, and minimally acknowledged, without the delay of updating the requested parts of the image, thus improving perceived responsiveness. This architecture allows some canceling of requests, by the network writer thread and the user input thread, if they can determine that the request is not needed, for whatever reason. There is still no reordering of requests, and every request that gets sent to the server is handled fully.

2.1.5 The Petra-Flow Framework

Petra-Flow [26] is a general framework for providing responsive behavior in the face of variable and volatile resource and service availability—precisely the environment we have with thinwire visualization. In this framework, a thread is spawned for each task, and the tasks are related to each other by a dynamic dependence graph. Task priorities are explicitly changed using the operating system facilities for setting thread priorities, and implicitly changed when the dependence graph changes. The dependence graph is changed when the user invokes a new task, or an old task completes. If a user action causes a task to become obselete—e.g., in a web browser, requesting a new page before the old page is finished loading—then whole branches of the dependence graph may become obsolete, and tasks on those branches will be canceled. All of these task coordination activities assume that the tasks are executing locally. Therefore, in a client-server situation, the Petra-Flow framework is limited to responsiveness on the client, and the server is seen as merely an unreliable resource. The responsive visualization architecture described in later in this chapter, while less general than Petra-Flow, is in another way more global, in that it incorporates the server and thinwire (i.e., the volatile resource) as *part* of the system, rather than external to it.

2.2 Design Constraints

Later in this chapter we will discuss the specific strategies and data structures that we have developed to ensure that the system operates correctly and with a high level of responsiveness. This section covers some of the many other aspects of the overall system design, aspects that may not themselves require elaborate

explanation or justification, but that still should be noted because they give context for the descriptions to come later. Our goal has been to formalize these aspects into a set of *constraints* that can be used later to prove certain properties of system performance. For example, we would ideally like to formalize a definition of responsiveness, and then prove that within certain constraints, the system is maximally responsive. This effort has not been entirely successful; nonetheless, we list below the constraints that we have identified, and in section 2.5 we provide an informal argument for the responsiveness of the system architecture.

2.2.1 Constraints Related to Problem Definition

These constraints are required, in the sense that if they were relaxed, we would be dealing with a different problem.

Constraint 1 *User and data are separated by a thinwire.*

The network connection is slow and unreliable; we cannot assume any minimum bandwidth.

Constraint 2 *The size of the dataset is very large.*

Specifically, it is too large to fit in main memory, and too large to be downloaded over the thinwire all at once (because it would take too long). An immediate implication is that data must be provided to the client dynamically, during the visualization session, rather than statically before the session begins. Another implication is that if the client keeps a local cache of data that it has downloaded (and it probably should), then the cache must be limited in size, and provision must be made for data to be removed from the cache.

Constraint 3 *The user can browse the data using multiple windows, and request priority is determined primarily by which window the request is for and secondarily by the request's priority among requests from that window.*

This seems reasonable, although other approaches are certainly possible—data elements might be assigned intrinsic importance values, independent of how or where the user views them. In our system, the window that has the current input focus has highest priority, and requests from that window are ordered by sequence number, with the most recent being highest priority.

2.2.2 Simplifying Constraints

These constraints are not required, they just represent the approach we have taken for now. They serve to simplify things, by reducing the parameter space from which we take our system design.

Constraint 4 *The user communicates through client software to a single server computer, which controls access to the data. The client and server computers support multi-threading, but need not have multiple processors.*

In other words, no distributed databases or peer-to-peer networks. There are three interface protocols that need to be designed: server-client, server-data, and client-user. Roughly speaking, the server-client interface is covered in this chapter, the server-data interface in chapter 3, and the client-user interface in chapter 4.

Constraint 5 *The server only sends data to the client if the data has been requested.*

No "push" semantics. Push capabilities might be beneficial, but they would be hard to implement with HTTP tunneling (see section 5.3).

Constraint 6 *The client makes a request to the server only if the request comes directly from the user.*

No prefetching. Section 5.1 discusses prefetching as it relates to our system.

Constraint 7 *All threads behave in a locally greedy manner.*

If a thread has work to be done, the thread will work on a task that is available, and not wait to see if more urgent tasks might soon become available. In our system, many of the threads get their tasks from a priority queue that is written to by other threads. When a thread finishes a task, if its priority queue is not empty, it will immediately start working on one of the tasks on it. An alternative might be something like this: if none of the tasks on the priority queue are important, then wait 500 msec before starting the next task, in the hope that a more important task will arrive in that time. (Of course this would only be beneficial if the average task completion time were larger than 500 msec.)

Constraint 8 *The number of threads is fixed, and the client makes only one connection to the server.*

We do not dynamically create threads. One method of HTTP tunneling, discussed in section 5.3, would involve dynamically creating threads.

2.2.3 Atomicity Constraints

We define a coarser granularity of tasks than what would be seen from an operating system perspective, and stipulate that tasks are atomic, in the sense that a thread that is working on a task must complete the task before starting another task. These are really simplifying constraints also, but it seems helpful to separate them into their own group.

Constraint 9 *The granularity of the range requests is fixed.*

The server makes one query to the database for each range request task; it cannot break requests down into smaller requests, or combine them into bigger requests. (Note, however, that each user request might be split into multiple range requests at the client side.)

Constraint 10 *Making one query to the database and retrieving zero or more blobs in response is a single task.*

In theory, a query could be interrupted by a more important request that comes along, but for simplicity we do not allow this.

Constraint 11 *Sending a single blob or other message across the network is a single task.*

A blob may not be broken down into pieces; if a high priority blob comes along to be sent to the client, it must wait until any pending action that is writing a blob to the network finishes.

2.2.4 Java Related Constraint

Because we want the client to run on a variety of platforms and be easily accessible on the Internet, we have chosen to implement it as a Java applet. This mandates the following constraint:

Constraint 12 *A single thread must handle both user input and graphical display output.*

The Java runtime environment provides the Event Handler thread for these operations, and though it is technically possible for another thread to write output to the screen, in such a case the interaction with the Event Handler thread could lead to inconsistencies. Therefore, we do not allow it.

2.3 Thread Architecture

Every aspect of our runtime system architecture is infused with the goal of providing maximal responsiveness. Figures 2.1 and 2.2 together show a simplified view of our runtime system architecture. The large rectangles represent threads. Most threads also have a small rectangle attached to them labeled "pq" or "q"; these represent priority queues or queues, the data structures that handle inter-thread communication and prioritizing of tasks. Solid arrows represent data flow (shown in figure 2.1), and dashed arrows represent commands (shown in figure

2.2). (The distinction between data and command may be somewhat arbitrary. For instance, the ranges that are sent to the server as part of a range request are shown as data, but they could also be considered commands.) Much of the complexity and sophistication of this architecture, for providing responsiveness, is encapsulated in the priority queues. Section 2.4 describes the priority queue structure and functionality in detail. Section 2.3.4 describes the behavior of each thread in detail. Here we look at the overview of system behavior.

Before going into detail, we give here a brief description of each thread:

GUI Event Handler Handles all mouse input and screen output.

Request Sender Sends requests from the client to the server; primary responsibility is writing the thinwire from the client side.

Renderer Controls the sending of screen output tasks (i.e., object rendering tasks) to the Event Handler. These tasks must be sent one at a time.

Searcher Pulls objects from the client side cache and sends them to the Renderer.

Response Receiver Primary responsibility is reading the thinwire for the client; reads blobs from the thinwire and sends them to the Unpacker.

Unpacker Uncompresses each blob, stores the geographic objects in the cache, and also gives the objects to the Renderer.

Request Taker Handles reading the thinwire on the server side; simply takes client requests and passes them on to the Data Provider.

Data Provider Manages the database and the server side cache; reads blobs from the database, stores them in the cache, and sends them to the Response Sender.

Response Sender Handles writing the thinwire from the server side; mainly just takes the blobs it gets from the Data Provider and sends them to the client.

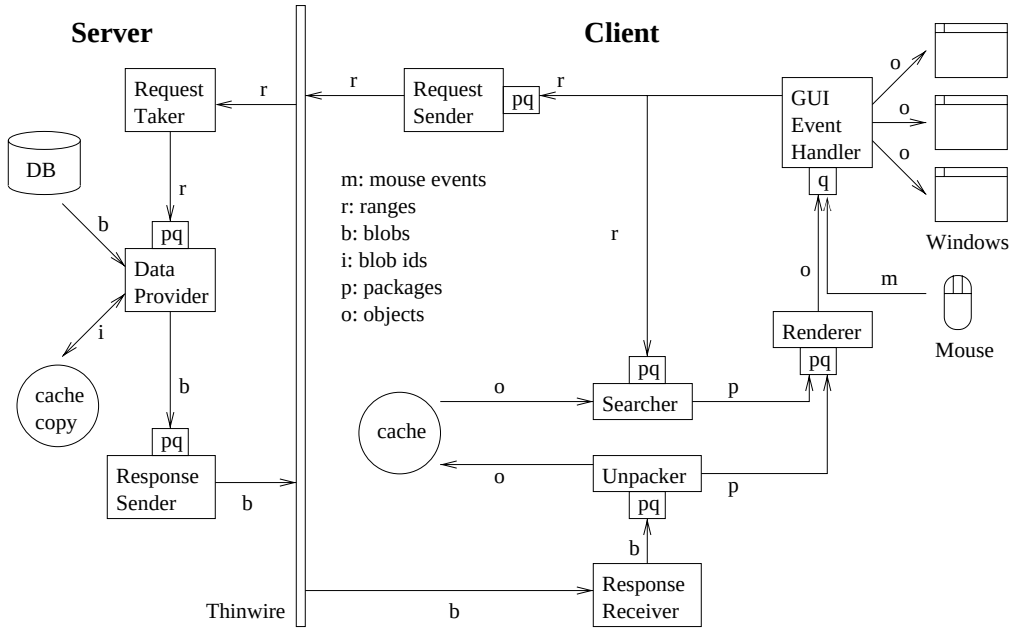

Figure 2.1: Runtime System Data Flow Diagram

2.3.1 Data Flow

The flow of data through the system is shown in figure 2.1. This diagram illustrates the normal activity of the system, which is retrieving geographic objects from the server and displaying them in response to user requests. Consider the sequence of events that follow from the user making a movement of the mouse, either for a pan or a zoom event:

1. The GUI Event Handler, upon receiving the mouse event, immediately makes some update to the screen, so that the user can see that the input was received and recognized by the system. The screen update will be either to translate the on-screen image (for a pan) or display a scaled version of the parent window's on-screen image (for a zoom or jump). This action is not represented in the diagram.

2. The Event Handler converts the pan or zoom event into one or more ranges and each range is sent in task form to the Searcher and the Request Sender threads. These two paths continue concurrently. Consider the path to the Searcher thread first; since that path does not go across the thinwire, it will likely complete sooner.

3. Upon receiving the range request task the Searcher pulls out from the cache all objects that intersect the range and puts those onto the Render schedule, in the form of a package (to be described in section 2.4).

4. The Renderer thread pulls out each object individually from the package and sends it back to the Event Handler.

5. The Event Handler displays each object to the window.

6. Returning to the second path that the range request takes, the Request Sender will simply write the request to the thinwire.

7. The Request Taker reads the range request from the thinwire and passes it, in task form, to the Data Provider.

8. The Data Provider first retrieves from the cache copy all blob ids for blobs that intersect the range and that the client already has, then it queries the database for any blob that intersects the range request and that is not in that set of blob ids.

9. The blobs retrieved from the database are added to the server side cache and passed to the Response Sender.

10. The Response Sender writes the blobs to the thinwire.

11. The Response Receiver reads the blobs from the thinwire and passes them to the Unpacker.

12. The Unpacker thread uncompresses each blob, producing individual objects. Each object is stored in the cache and put into a package, and the package is sent to the Renderer.

13. The Renderer sends each object from the package to the Event Handler.

14. The Event Handler displays each object that it receives.

For clarity, one dataflow path has been left out of the diagram. When the system shows feature labels, the Event Handler thread will directly search the cache to find what feature labels are near the current mouse position. This has implications for concurrent access to the cache data structures, as will be discussed in section 2.8.4. It also introduces a small quirk in the system: on occasion, the Event Handler will retrieve and display a feature label for an object

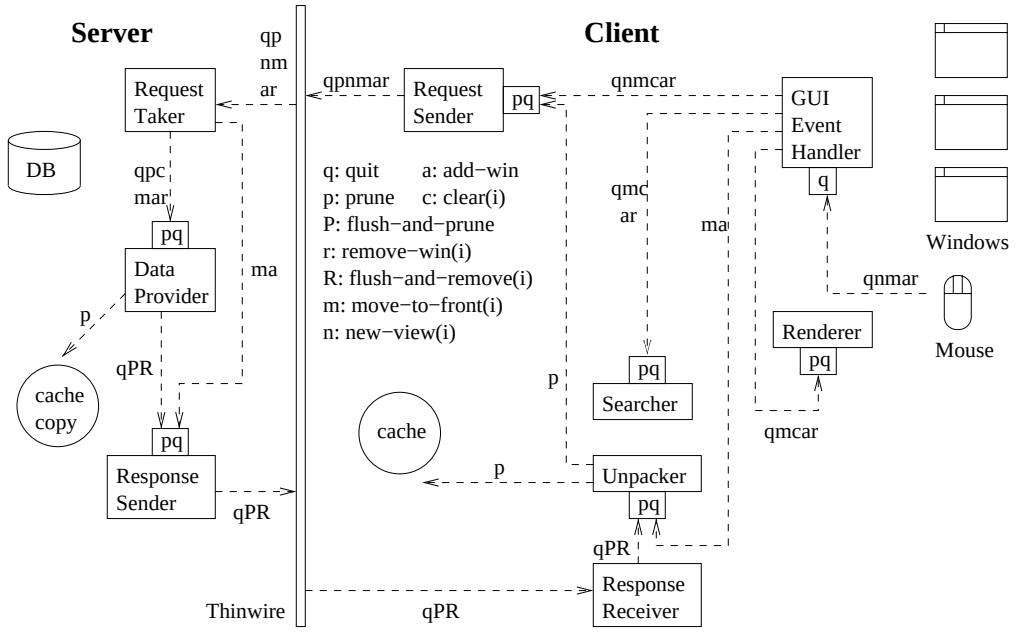

Figure 2.2: Runtime System Command Flow Diagram

before it has rendered the object itself. However, the Unpacker thread puts all new objects onto the Render Schedule as well as into the cache, so any object whose label is being displayed is at least on schedule to be rendered shortly. In practice we have not found this quirk to be problematic.

2.3.2 Command Flow

The command flow diagram, figure 2.2, includes three different command forms, but does not explicitly distinguish among them. A command may be a *mouse event*, such as a menu item selection or a button push; a *task* that is placed on a priority queue; or an *operation* that is executed on a priority queue, the cache, or the cache copy. Operations on priority queues change their internal structure, and possibly remove tasks from the queue. As an example, when the user pushes a button to close a window, that mouse event is put onto the Event Handler's queue, and in figure 2.2 this is represented by the label "r" on the arrow from the mouse to the queue. The Event Handler then (among other things) puts a remove-win task onto the Request Sender's priority queue and executes the remove-win operation on that priority queue to make internal changes reflecting the removal of the window. In figure 2.2, both of these command forms are

represented by the single "r" label on the arrow from the Event Handler to the Request Sender's priority queue.

What follows is a list of commands, each with a description of its purpose, what forms it takes, and how it flows through the system. More details about the behavior and internal structure of the priority queues is in section 2.4; more about the mouse events is in chapter 4.

quit The quit command is initiated by a menu item selection event being passed from the mouse to the Event Handler. From there, it is converted into a task that travels throughout the system to all threads. All threads cease execution upon receiving a quit task; some threads first pass the quit task on to another thread (or three other threads, in the case of the Event Handler). The quit task is sent directly from the Event Handler to the Searcher, Renderer, and Request Sender threads; the Response Receiver and Unpacker threads have to wait until the task comes back from the server side. This delay is not strictly necessary, it just ensures that the socket connection will be cleanly closed by both sides. If for some reason the quit command never comes back, the Response Receiver and Unpacker threads will timeout and exit.

add-win To open a new window, the user initiates the add-win command with a mouse click, which event is caught by the Event Handler. The result that we want is to have every priority queue in the system execute the add-win operation, which will update its internal structures to account for the new window. The Event Handler may directly execute the add-win operation on the priority queues on the client side. For the priority queues on the server side, the Event Handler must send an add-win task to the Request Sender, which passes it across the thinwire to the Request Taker, and the Request Taker can then directly execute the add-win operation on the priority queues on the server side.

remove-win(i) and flush-and-remove(i) To close the i-th window, the user clicks the close icon on that window, and the mouse event, which we call remove-win(i), is sent to the Event Handler. At this point, the remove-win command takes a somewhat different path from the add-win command, because of the necessity of maintaining consistency between the cache and the cache copy. The Event Handler executes update operations on three of the priority queues on the client side—all but the Unpacker's priority queue. One result of these internal updates to the priority queues is that

all pending tasks for the closed window are discarded. The Event Handler also sends a remove-win task to the Request Sender, which forwards the task to the server's Request Taker. The Request Taker executes the remove-win(i) operation on the Data Provider's priority queue, discarding any pending tasks for that window, and puts a remove-win task onto the queue. When the Data Provider receives the remove-win task, it executes a flush-and-remove(i) operation on the Response Sender's priority queue, and puts a flush-and-remove(i) task onto that queue. The effect of the flush-and-remove(i) operation is to fully process—not discard—all pending tasks for the i-th window and *then* update the queue internal structure to reflect the removal of the window. The flush-and-remove(i) task is passed across the thinwire to the Response Receiver, which executes the flush-and-remove(i) operation on the Unpacker's priority queue.

move-to-front(i) The system is designed to give highest priority to actions that are for the window that currently has the input focus—usually the window that was clicked on most recently. Information about the relative priorities of the windows is stored in the priority queues. When the user clicks on a window, we want to execute the move-to-front(i) operation on each of the priority queues in the system, indicating that the window that clicked on should now be at the front of the window priority list. The move-to-front(i) command moves through the system in exactly the same way as the add-win command: priority queues on the client side can be updated immediately by the Event Handler, but a task must be sent across the thinwire to the Request Taker before the priority queues on the server side can be updated.

new-view(i) and clear(i) In the case of this command, we have given different names to the different forms: new-view(i) stands for both the mouse event and the task, while clear(i) represents the associated priority queue operation. When the user does a zoom or a jump to a new location, the system redraws the entire window; therefore, many of the outstanding tasks for that window can be discarded, subject to the constraint of maintaining cache consistency. The pattern is similar to that for remove-win(i). When the Event Handler catches the associated mouse event, it executes clear(i) on three of the client side priority queues—all but the Unpacker's queue. clear(i) causes all tasks for the i-th window to be discarded from the priority queue. A new-view(i) task is sent to the Request Sender, and then across the thinwire to the Request Taker, at which point the clear(i) operation is

called on the Data Provider's priority queue. The priority queues for the Response Sender and the Unpacker are never affected, because any data that has already been sent from the Data Provider, and therefore noted in the server's cache copy, must complete its journey into the client side cache.

prune and flush-and-prune The prune command is the only one that does not originate with a mouse event. The Unpacker keeps track of the number of points downloaded (i.e., line endpoints and shape points), and after every N points, where N is a startup parameter, it initiates a prune command by putting a prune task onto the Request Sender's priority queue. From there the task is passed to the Request Taker, which puts it onto the Data Provider's queue. When the Data Provider gets the prune task, it first executes the prune operation on the cache copy. At this point we need to be careful, because in order to maintain cache consistency, any blobs that are noted in the cache copy before the prune operation on the server side must also be stored in the cache before the prune operation on the client side, and any blobs after the prune on the server must also be after the prune on the client. What we want, therefore, is to fully process all pending tasks on the Response Sender queue and then immediately process the prune task, and the same thing on the Unpacker queue. This is what the flush-and-prune command does. So the Data Provider next executes the flush-and-prune operation on the Response Sender priority queue, and the Response Sender sends a flush-and-prune message across the thinwire back to the client, whereupon the Response Receiver executes the flush-and-prune operation on the Unpacker priority queue. Finally, upon receiving the prune task the Unpacker executes the prune operation on the cache. The necessity for this indirection (Why not just have the Unpacker prune the cache to begin with?) is explained in section 2.6.

2.3.3 Note on the Event Handler Thread

The queue that the Event Handler uses for its input is part of the Java runtime environment, and so we do not have the ability to replace it or modify it; we can only add tasks for the Event Handler thread onto it. Since the Event Handler gets both display tasks and mouse input tasks from the same queue, we need to be careful about how we use that queue. If we have a large number of objects to render, and we put them all onto the queue, then any new mouse event that occurs will have to wait until all of the objects have been rendered before it can be handled. This is obviously bad for responsiveness. Fortunately, Java provides

the option of putting a task onto the queue and blocking until that task has been completed. Using this option, we can guarantee that only one render-feature task will be on the queue at a time, so any new mouse event can be handled quickly. In general, any task that the Event Handler performs should be as quick as possible, for responsiveness; thus, in our system we try to offload as much work as possible from the Event Handler onto the other threads.

2.3.4 Thread Behaviors

This section repeats much of the information in sections 2.1 and 2.2, but this time looking at it from the perspective of each thread, rather than the perspective of the data or commands. Much of the logic for responsiveness, particularly the prioritizing of tasks, is encapsulated in the priority queues, so the thread behaviors tend to be quite simple. (The one exception is the Event Handler thread.) In the following, the action of each thread is listed with the input that triggers the action. The input comes from a queue or priority queue, the network, or the mouse. The threads are discussed in the order that they might be traversed by following the path of actions resulting from a single mouse event.

Event Handler Thread

The Event Handler thread is provided by the Java runtime environment to handle all input and output that goes through the graphical user interface. It gets its tasks from a queue that is also provided by the runtime environment. The GUI automatically puts mouse input events onto the queue, and other parts of the system are also allowed to put events, in particular display output events, onto the queue.

Several of the actions listed below make use of the *update-view* subroutine, which redraws the window contents from scratch. It includes these steps:

1. call the clear(i) operation on the Searcher and Renderer priority queues;

2. add a new-view(i) task to the Request Sender priority queue; and

3. form range request tasks for nine equal sized rectangular parts of the screen, giving highest priority to the center rectangle and lowest to the corners, and put those tasks onto the Searcher and Request Sender priority queues.

Here are the input tasks and events, and their associated actions:

- *mouse drag(dx,dy)*: translate the current screen image by (dx, dy), form range request tasks for the newly opened rectangles in the x and y directions, and put those tasks onto the Searcher and Request Sender priority queues.

- *zoom*: replace the current screen image with a scaled version of itself, and call *update-view*.

- *jump*: replace the current screen image with a scaled version of the appropriate rectangle from the parent window's screen image, and call *update-view*.

- *add-win*: create a new window in the GUI, initially showing a scaled version of the appropriate rectangle from the parent window's screen image; call the add-win operation on all client side priority queues; send an add-win task to the Request Sender thread; and call *update-view*.

- *remove-win*: close the window; call the remove-win operation on the priority queues for the Request Sender, Searcher, and Renderer; and send a remove-win task to the Request Sender.

- *change input focus window*: call the move-to-front operation on all priority queues on the client side, and send a move-to-front task to the Request Sender.

- *geographic object*: render the object to the screen.

- *quit*: put a quit task onto the priority queues for the Searcher, the Renderer, and the Request Sender. The Event Handler thread is controlled by the Java runtime environment, so we don't need to explicitly stop it.

Searcher Thread

The Searcher is responsible for pulling objects out of the cache. It gets its tasks from its priority queue.

- *range request*: search the cache for all polygons, polylines and landmarks that intersect the range, package them up into a queue with the polygons first, and add the package to the Renderer's priority queue.

- *quit*: exit.

Renderer Thread

The Renderer is responsible for getting objects displayed on the screen. "Renderer" is a bit of a misnomer, because the actual screen rendering is done by the Event Handler thread. (This is required, or at least highly recommended, by the Java runtime environment.) But the Renderer prepares each object for rendering, and controls the flow of display tasks to the Event Handler. It gets its tasks from its priority queue.

- *geographic object (polygon, polyline or landmark)*: prepare the object for rendering, by scaling or translating its coordinates, if necessary; add a geographic object task to the Event Handler's queue; and block (wait) until the rendering has been completed by the Event Handler thread.

- *quit*: exit.

Request Sender Thread

The Request Sender is responsible for writing the thinwire. It gets its tasks from its priority queue.

- *range-request, new-view(i), add-win(i), remove-win(i), move-to-front(i), prune*: simply pass the input task across the thinwire to the server's Request Taker.

- *quit*: exit.

Request Taker Thread

The Request Taker's primary responsibility is to read from the thinwire; it also executes some operations that are to trivial to require a separate thread. It gets its input from the thinwire.

- *range-request, prune*: send the task to the Data Provider's priority queue.

- *new-view(i)*: call the clear(i) operation on the Data Provider's priority queue.

- *add-win, move-to-front(i)*: execute the add-win or move-to-front(i) operation on both server side priority queues;

- *remove-win(i)*: execute the remove-win(i) operation on the Data Provider's priority queue, and add a remove-win(i) task to that queue.

- *quit*: put the quit task onto Data Provider's priority queue, and exit.

Data Provider Thread

The Data Provider's main responsibilities are in searching the database and managing the cache copy. It gets its tasks from its priority queue.

- *range request*: search the cache copy for blob ids of blobs that intersect this range and that have already been sent, query the database for any other blobs that intersect this range, put the blob ids for these new blobs into the local cache, and put a blob task for each new blob onto the Response Sender's priority queue.

- *prune*: prune the cache copy (see sections 2.6 and 2.7 for details about the pruning process), execute the flush operation on the Response Sender priority queue, and add the prune task to that queue.

- *remove-win(i)*: execute the flush-and-remove(i) operation on the Response Sender priority queue, and put the remove-win(i) task onto that queue.

- *quit*: put the quit task onto the Response Sender priority queue.

Response Sender Thread

The Response Sender is responsible for writing to the thinwire. It gets its tasks from its priority queue.

- *blob*: write the blob to the thinwire.

- *remove-win(i)*: send the remove-win(i) task across the thinwire to the Response Receiver.

- *prune*: send the prune task across the thinwire to the Response Receiver.

- *quit*: send the quit task across the thinwire to the Response Receiver, close the socket connection, and exit.

Response Receiver Thread

The Response Receiver is responsible for reading the thinwire. It gets its tasks from the thinwire.

- *blob*: read the blob from the thinwire and send it to the Unpacker thread.
- *prune*: execute the flush operation on the Unpacker priority queue, and add the prune task to that queue.
- *remove-win(i)*: execute the flush-and-remove(i) operation on the Unpacker priority queue.
- *quit*: add a quit task to the Unpacker priority queue, close the socket connection and exit.

Unpacker Thread

The Unpacker's main responsibility is to unpack objects from the blobs and send those objects to the cache and the Renderer. It gets its tasks from its priority queue.

- *blob*: uncompress the blob and pull out individual geographic features; store those features in the cache; package them up into a queue, with the polygons first; add the package to the Renderer priority queue; note the number of points that have been downloaded, and if necessary initiate a pruning by adding a prune request to the Request Sender priority queue.
- *prune*: prune the cache, as described in section 2.7.
- *quit*: exit.

2.3.5 Discussion

An attempt has been made to assign tasks to threads in such a way that each thread spends most of its active time (i.e., time when it is not blocked, waiting for a task) in a single activity that can be done concurrently with other threads' activities. For example, on the server side the main activities for the three threads are reading from the thinwire, querying the database, and writing to the thinwire, respectively. The concurrency between threads is usually pipelined concurrency as, for example, when the Data Provider is working on one range request while

the Request Taker is reading the next one from the thinwire. Full concurrency happens when both the Request Sender and the Searcher are working on the same range request.

One thing to notice is that the Request Taker and the Response Receiver are simple threads that exist mainly to make the thinwire seem transparent to the rest of the system. In other words, if there were no thinwire—if the client and server were on the same machine—then the Request Sender would directly access the priority queues for the Data Provider and the Response Sender, and the Response Sender would directly access the priority queue for the Unpacker. But the thinwire imposes the restriction that all messages passing between the client and the server, in either direction, must be handled in first-in first-out order. The Request Taker and Response Receiver threads allow these messages to be reordered according to our priority queue order, as far as the rest of the system is concerned.

It might seem at first glance that this architecture contains more threads than are necessary. For example, why not eliminate the Response Sender and have the Data Provider write the blobs directly to the network? The answer is twofold. First, the priority queue that comes with the Response Sender is itself an important element for system responsiveness. Without that priority queue, the blobs that are retrieved from the database would have to be sent across the thinwire in first-come first-served order. But the thinwire is assumed to be relatively slow as compared to other parts of the system, especially for transmitting blobs, which are larger than other system messages. Therefore, if requests come in bunches, which is expected, and writing blobs to the thinwire is a bottleneck, then if the blobs cannot be reordered, it will happen quite frequently that higher priority blobs will wait while lower priority blobs are sent first.

The second reason is that having separate threads allows for more parallelism. Since the thinwire is slow, or may even go down temporarily, we expect that the Response Sender will spend a significant amount of time waiting in a blocked state. During that time the Data Provider thread can continue searching the database and getting blobs ready for transmission (assuming, of course, that multiple requests are waiting on the Data Provider queue).

2.4 The Priority Queues

The priority queues are an essential component of the responsive system architecture, providing two primary functions: inter-thread communication and task scheduling. Priority queue public methods are declared with the Java key-

word `synchronized`, which means that if one thread is executing a synchronized method for an object then no other thread may execute a synchronized method for that object. So the priority queues implement a form of the producer-consumer problem, with the low level synchronization provided by the Java runtime environment. The prioritization requirements are these:

1. Most tasks, such as range requests, are associated with a particular window. These tasks are thus partitioned into priority classes, one per window. If priority class i has higher current priority than priority class j, then all tasks in class i have higher priority than all tasks in class j.

2. The relative priorities of the priority classes is changed dynamically by "move-to-front" operations. When a user sets the input focus to a window, the priority class associated with that window is moved to the front of the priority list, and all other priority classes maintain their previous ordering.

3. Within a priority class, tasks are prioritized according to a sequence number, where higher sequence numbers are given higher priority. Tasks are not in general added to the schedule in sequence number order. (If task T_1 has a higher sequence number than task T_2, it means that T_1 is generated by a user request that is "not less recent" than the request that generated T_2. Usually it means "more recent", but in some cases multiple tasks are generated at the same time but assigned different sequence numbers to accommodate other priority considerations. In particular, when a user zooms or jumps to a new location, the screen is split into nine rectangles, and the middle rectangle is given the highest sequence number, the corners the lowest, etc.)

4. Some tasks, such as *quit* and *prune* are not associated with a window. These are called "global" tasks, and have higher priority than all other tasks. Global tasks should be serviced in FIFO order.

The remainder of this section describes the structure and behavior of the priority queues in detail. Section 2.4.1 describes the basic behavior, which is used by the Request Sender, Searcher, and Unpacker priority queues. The Renderer, Data Provider, and Response Sender each need priority queues with additional functionality. The remaining three subsections cover each of those queues—why the basic priority queue is insufficient, and what the additional features are.

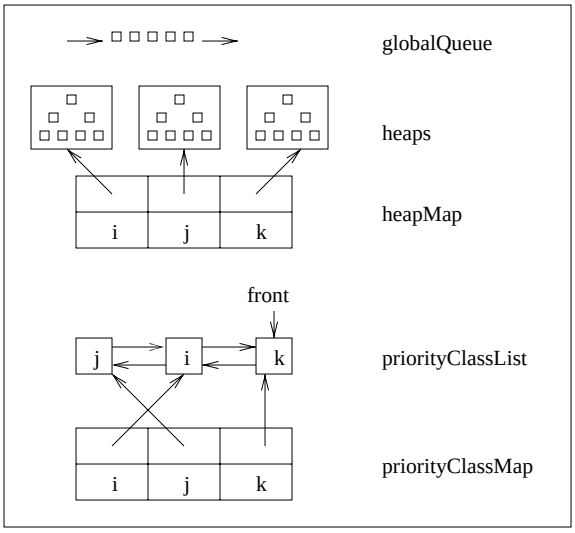

Figure 2.3: The Basic Priority Queue

2.4.1 The Basic Priority Queue

Figure 2.3 shows the internal structure of the basic priority queue. Each priority queue includes the following data structures:

- `globalQueue`: an ordinary queue, for the global tasks.
- Several heaps, one for each priority class. Each heap stores the tasks for that priority class, heap-ordered by sequence number.
- `priorityClassList`: a move-to-front list, implemented as a doubly linked list, where each node stores a priority class id.
- `priorityClassMap`: a hash table mapping priority class ids to nodes in the move-to-front list, so that move-to-front can be accomplished in constant time.
- `heapMap`: a hash table mapping priority class ids to heaps, so that `addTask` can be accomplished in time that is independent of the number of priority classes.

Following is a list of the priority queue supported operations. Each of these is executed as an atomic operation, so only one of these operations can be performed on the priority queue at a time. Therefore, it is important that these operations

be as fast as possible. In the following, let n be the number of priority classes, and let m be the size of the largest heap at any given time.

- `addPriorityClass(id)`: add a new priority class to the front of the move-to-front list, giving it the highest priority. Called when the user adds a new window. Requires constant time.

- `removePriorityClass(id)`: remove a priority class from the move-to-front list, and remove all tasks associated with it. Called when the user deletes a window. Requires constant time.

- `moveToFront(id)`: move the given priority class to the front of the list. Called when the user clicks on a window, giving it the input focus. Since the list is implemented as a linked list, and the node that we want to move can be found in constant time with a hash table, requires constant time.

- `getNextTask()`: return the highest priority task, and remove it from the schedule. If the global task queue is non-empty, then return the front task on that queue, else start searching through the heaps, in priority class order, and return the top task on the first non-empty heap. Requires time $O(n + \log m)$

- `addTask(priorityClassId, sequenceNumber, task)`: add the given task to the appropriate heap, with the given sequence number. Requires time $O(\log m)$

- `addGlobalTask(task)`: add a task to the global task queue. Requires constant time.

- `clear(priorityClassId)`: clear the heap associated with the given priority class id. Called when a user does a new-view operation, such as zoom or jump to a new location. Requires constant time.

- `flushToGlobalAndRemove(priorityClassId)`: move all tasks for a given priority class to the global task queue, in heap order, and then remove the priority class. This is the flush-and-remove(i) command mentioned above. Requires time $O(m \log m)$.

- `flushToGlobalAndAdd(task)`: move all tasks for all priority classes to the global task queue, in order of priority (i.e., first consider the priority class, then the priority within the class), and then add the given task to the (back

of the) global task queue. This is used for the flush-and-prune operation. Since it is an atomic operation, and since tasks on the global queue take precedence over those on the heaps, the prune is guaranteed to be processed after any previously added tasks and before any tasks that will be added in the future. Requires time $O(nm \log m)$

Under normal operation there will be no global tasks, and a thread that executes `getNextTask()` will get the highest priority task for the window with the current focus, which is the ideal for responsiveness. The `flushToGlobalAndAdd` and `flushToGlobalAndRemove` methods cause a temporary suspension of the normal prioritizing, but since these operations are relatively rare we accept this occasional reduction in responsiveness. And for the same reason we can accept the relatively long running time of these two operations.

The next sections describe three situations where the basic priority queue needs to be extended with additional functionality.

2.4.2 The Renderer Priority Queue

The sequence numbers that we use to prioritize tasks within a priority class are associated with user requests—zooms, pans, etc.—and normally there is one task per sequence number. Even if a single user request is divided into multiple range requests, each range request is given a unique sequence number and associated with a single task. But when it comes to the Renderer, the association of one range request to one task and one task to one sequence number must break down because of the interaction of the following considerations:

1. For display, we should send only one geographic object at a time to the Event Handler (recall the discussion from section 2.3.3 about how it is important to send only fast tasks to the Event Handler, and only one at a time).

2. Each user request results in the need to display multiple geographic objects.

3. At the time that the sequence numbers are assigned, when the user makes the request, we do not know how many objects are going to need to be displayed for this request, either from the cache or from the server.

4. Among the objects associated with a single request, there are restrictions on the order in which the objects must be displayed. The data will be discussed more fully in chapter 3, but in summary a polyline that is within

a polygon must be drawn after the polygon, or else it would be covered up when the polygon is drawn. Similarly for landmarks or smaller polygons that are within the main polygon.

5. In a priority queue, if two tasks are in the same priority class and have the same sequence number then the order in which they will be processed is unspecified.

So the dilemma is this: if the Searcher and Unpacker (the two threads that write the Renderer queue) put individual objects onto the queue as separate tasks, each with the same sequence number (the one associated with the request that they come from), then the objects might be displayed in the wrong order, but if they package all the objects together into one task then that task would be too long for the Event Handler and would have a negative impact on responsiveness. We cannot assign individual sequence numbers to each object, because at the time that this would need to happen, when the request is made, we would not know how many sequence numbers to "reserve", but we cannot disassociate the object sequence numbers from the request that picked them because then we would lose the prioritization based on user actions.

Our solution to this dilemma is to add functionality to the priority queue that allows a "package" of tasks to be treated as a unit *within* the priority queue, but to allow an individual task to be removed from the package when `getNextTask()` is called. Figure 2.4 shows what a single heap would look like on this priority queue. Each heap node contains a simple queue, and each node on that queue is a task. The `getNextTask` method removes the first task from the queue that is at the top of the heap, but does not remove that top heap node unless its queue is empty. It is acceptable for a heap node at the top to be moved down the tree when a higher priority heap node is added, even if it (the node previously at the top) has had some of its tasks removed. The only restriction is that tasks within a heap node are removed in *order*; they do not need to be removed in *sequence*.

With this solution, all of the above considerations are satisfied: only one geographic object is sent to the Event Handler at a time, the objects are sent in the correct order, and each of these objects is correctly prioritized according to its request's sequence number.

2.4.3 The Response Sender Priority Queue

There is a different problem with the Response Sender priority queue that also results from the interaction of a number of considerations. Consider the following:

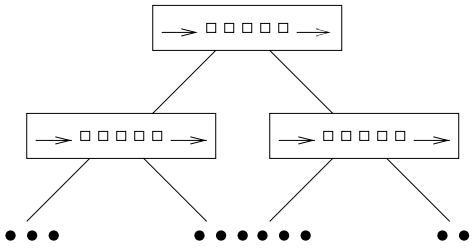

Figure 2.4: A Heap on the Renderer Priority Queue

1. A single blob will in general intersect multiple range requests.

2. Once the Data Provider has sent a blob to the Response Sender, it will not send that same blob again (assuming the blob has not been pruned form the cache in the interim).

3. It is possible for a task to remain on a priority queue while any number of higher priority tasks come and go.

Now suppose the Data Provider thread gets a range request R_1 that intersects blob B, which is sent to the Response Sender. Before B is removed from the Response Sender queue, the Data Provider thread gets a new range request, R_2, that also intersects B and that has higher priority than R_1. Since the Data Provider has already sent B, it will not be included in the response for R_2. The priority of B has effectively been limited to the priority of R_1. If several more range requests come along with even higher priorities and also intersecting B, then B will sit on the queue while responses that *should* include B pass it by. B's "demotion" to a lower priority than the user wants becomes even more egregious. For responsiveness, B should really be *promoted* to the highest priority of any range request that intersects it. The extended functionality of the Response Sender priority queue allows this to happen.

When blob B intersects both range requests R_1 and R_2, there are two possibilities: either R_1 and R_2 are in the same priority class, or they are not. In the first case, we want to associate B with whichever request has the higher priority. In the second case, the situation is not so clear, because the relative priorities of R_1 and R_2 may change whenever the user gives input focus to a different window. Therefore, in the second case, we associate B with *both* requests in the priority queue, but make sure that it only gets sent to the client once. The rest of this section describes the necessary changes to the priority queue.

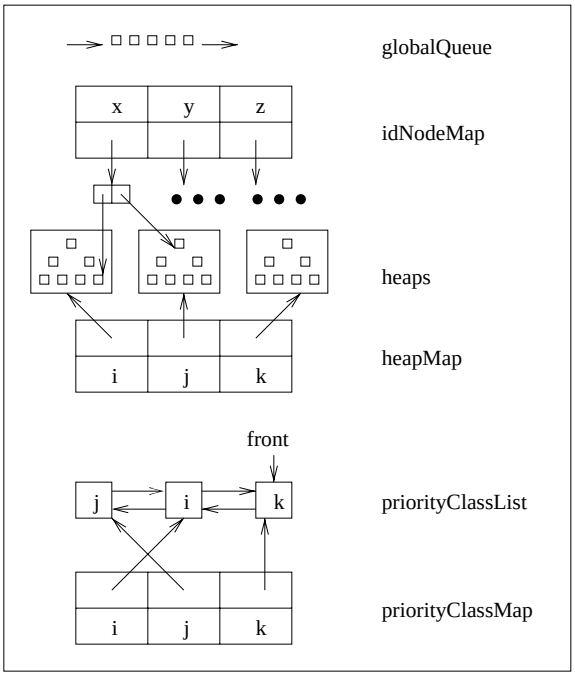

Figure 2.5: The Response Sender Priority Queue

Figure 2.5 shows the priority queue with one more data structure, `idNodeMap`, which maps each blob id to a list of heap nodes that contain (a reference to) that blob. (To avoid clutter, the figure only shows the full structure for the blob with id x.) To take this new data structure into account, we must redefine `addTask()` and `getNextTask()`, and define a new method, `upgrade(id, priorityClassId)`.

- `addTask()`: the only change is that for each blob added we must add an entry to `idNodeMap`, which is a constant time addition.

- `getNextTask()`: if the task is a blob task, the additional work is to iterate through the list of heap nodes associated with the blob and *invalidate* the blob in that node (essentially, set it to NULL), and to remove the blob's entry from `idNodeMap`. (Of course the blob is not invalidated in the current task that we are processing.) Since there can be at most n nodes in the list, and removing an entry from a hash table takes constant time, the time for `getNextTask` is still $O(n + \log m)$.

- `upgrade(id, priorityClassId)`: look for the blob id in `idNodeMap`. If it is not there, then do nothing (the blob has already been sent to the client). If it is there, then get the list of heap nodes associated with the blob and see if the blob is already in the appropriate heap (the one with the right `priorityClassId`). If so, update the node's priority in that heap, and "bubble" it up to its proper place in the heap. If not, add the blob to the appropriate heap, and add a reference to this new heap node in the list of nodes for the blob in `idNodeMap`. There are at most n nodes in the list, and adding to a heap takes $O(\log m)$ time, so the time for `upgrade` is $O(n + \log m)$.

To account for this special behavior of the Response Sender queue, we must modify the Data Provider's behavior, as specified in section 2.3.4. When receiving a range request, and after searching the cache copy for blob ids of blobs that have already been sent to the Response Sender, it must execute the upgrade(id) operation on the Response Sender queue for each of those blob ids.

It is possible that the upgrade functionality would also benefit the Response Receiver queue. We have not implemented this functionality because (1) it would require an extra message to be sent across the thinwire, and (2) we assume that the thinwire is the primary performance bottleneck, which means that once a blob has made it across the thinwire, it is likely to be processed quickly on the client side, relative to the time needed to send a single message across the

thinwire. Testing would be required to decide if this is a valid assumption; section 5.1 discusses the difficult issues in setting up a reliable testing infrastructure for responsive thinwire systems.

2.4.4 Data Provider Priority Queue

For most purposes, the client does not need to keep track of which responses are for which requests, or even if a request was responded to at all; it can simply send requests and display whatever data it receives. If the only blobs that intersect a particular range request are ones that the client already has, then as far as displaying correct maps, the server shouldn't need to respond to that request at all. But it is very helpful for the user to know whether or not the client is currently waiting for data from the server. In other words, "Is nothing new being drawn to the screen because the system is stalled, or because there is nothing left to draw in this region?" To answer this question, the client program includes a small "light" on the display that is on whenever there are outstanding requests at the server. But in order for this to work we do need to get a response to every request, even if the response has no content.

In figure 2.2, the Request Taker executes a clear(i) operation on the Data Provider queue when it receives a new-view(i) command. This is not exactly correct, because if range requests were cleared out of the Data Provider queue, those requests would never be responded to. Instead, the Request Taker sets a bit flag in every range request in the i-th heap to indicate that the request is no longer valid. When the Data Provider pulls that request off of the queue, it will simply send an empty response back to the client. The priority queue must, therefore, provide an additional operator, `invalidate(i)`, which invalidates each task in heap(i). This operation takes $O(m)$ time.

2.5 Responsiveness

The system architecture is designed for responsiveness in several ways: (1) The Renderer only puts one task onto the Event Handler queue at a time, and only one geographic object is rendered in each task, so that the event handler never has to be excessively delayed in giving the immediate acknowledgment—the 0.1 second level of interaction—to a user input. (2) The priority queues are designed to give highest priority to tasks for the window with input focus, and to the most recent among all the tasks for a window. (We have to assume that the window with input focus is the one that the user is most interested in.) (3) The

concurrency that results from separating responsibilities into separate threads, so that most threads have only one primary responsibility, ensures that, as much as possible, slow tasks do not hold up fast tasks, and low priority tasks do not hold up high priority tasks.

Now we would like to evaluate these techniques, to get some idea of how good the responsiveness of the system is. First, in section 2.5.1, we make some theoretical arguments based on our knowledge of how the system works, and then in section 2.5.2 we describe some rudimentary experimental testing.

2.5.1 Theoretical Arguments for System Responsiveness

We would like to be able to say that the system is optimally responsive, but this is clearly not possible—there are so many alternative design possibilities that we cannot be certain that the design we have chosen is the best. It would be nice to say that the Event Handler always gives an acknowledgment to user input within 0.1 seconds, but we can't say that either because it might take longer than that for it to finish rendering an object that it is working on when the request arrives. We would like to say that no high priority task ever has to wait for a low priority task, but that also isn't true because, among other things, a thread might be working on a lower priority task when the high priority task arrives. Instead, we settle for the following two claims, which are highly qualified, but which still shed some light on how the system works, and why it is responsive.

Claim 1 *When the user makes a mouse input, the initial system acknowledgment of receipt of the input cannot be delayed by any more than (1) the rendering of a single geographic object; and (2) the acknowledgment of any previous mouse inputs that have yet to be acknowledged, which does not involve handling any geographic objects.*

Justification: The Renderer thread is coded so that once it has put an object onto the Event Handler queue for rendering, it will not put another object onto the queue until the rendering of the first object is finished. So no more than a single geographic object rendering task can be ahead of any given mouse input task on the Event Handler queue. Since the Renderer and the mouse are the only players that put tasks onto the Event Handler queue, the only additional tasks that a mouse input will have to wait for are previous mouse inputs. But the only things the Event Handler does in response to a mouse input are direct manipulations of the screen and putting tasks onto priority queues, none of which involves the handling of any geographic objects. □

In fact, it should be extremely rare for a mouse input to have to wait for previous mouse inputs. The user would have to make multiple mouse actions in rapid succession, and even then the Event Handler would probably not get backed up, because it is designed to handle mouse events quickly.

In the next claim we look at how much a task is delayed by other tasks. There are two types of delay that, for the purposes of this claim, we consider *necessary* delays. (1) When a task arrives to the priority queue of a thread, that thread might be currently performing another task, such as writing the network. In our system, waiting until the thread is done with that task is considered a necessary delay. (See the atomicity constraints in section 2.2.3. (2) The time that it takes to add a task to a priority queue is a necessary delay. This time is logarithmic in the number of tasks that are already on the particular heap that the task will be added to within the queue. Of course these delays are not strictly necessary in an absolute sense, since it is possible to imagine an architecture in which they would not occur, but in our architecture they are unavoidable, and considering them to be necessary helps to illuminate the workings of the architecture, which is the purpose of these claims.

Claim 2 *In the absence of prune commands, the blobs associated with a range request will arrive to the client without being delayed any more than is necessary, as defined above.*

Justification: For blobs that are already at the client, we are done, so consider the path that a request takes to the server and its response blobs take back to the client. From the Event Handler, where the request is initiated, to the Data Provider, the task associated with the request passes through two priority queues and two threads. At the Request Sender priority queue, the structure of the queue guarantees that the new task will jump in front of any lower priority task on the queue, so the only delays could be to wait while the Request Sender finishes a previous task, and to wait while the task is being added to the priority queue, which are both necessary delays. Similarly for the Data Provider queue. The Request Sender and Request Taker simply forward the task along the path before going on to the next task. At the Data Provider thread there are two possibilities for any blob that is required: either the required blob has already been sent on to the Response Sender, or it has not. If not, the blob is pulled from the database and put on the Response Sender queue with the current request priority, and the same argument as for the other two queues applies to this one. If the blob has already been sent on, then it may be on the Response Sender queue as part of a task that has lower priority. In this case the Data Provider will execute the `upgrade` command, so the blob will then have the higher priority,

and again the same argument applies. □

Note that there are two significant qualifications that limit the impact of this claim.

1. It only applies "in the absence of prune commands", because of the command `flushAndPrune`, which executes outside the normal prioritization rules. This is required for maintaining cache consistency (see section 2.6). Since we assume that pruning is a relatively rare event, this should not have a significant impact on perceived responsiveness.

2. The responsiveness is only guaranteed up to the point that the data arrives at the client; it does not extend to unpacking the data and displaying it. The reason is that we do not execute the `upgrade` command on tasks that are on the Unpacker priority queue, so it is possible that a blob could languish on that queue with a lower priority than it should have, if higher priority requests come by that would have included it. As discussed above, we expect that the thinwire is the performance bottleneck in the system, so that once a blob arrives at the client it should be processed relatively quickly, as compared to the time that would be needed to send an `upgrade` message across the thinwire.

Notice that the delay for adding a task to a queue should be quite small, since the dependence is logarithmic, and the `addTask` operation is performed in memory, whereas each thread along the path from the client to the server and back is primarily occupied with I/O: the Request Sender and Response Sender write the thinwire, and the Data Provider reads the database. If there are tasks previously on a particular priority queue, then it is likely that the thread associated with that queue will be occupied with an I/O operation during the `addTask` operation (which is performed by a different thread). Since I/O operations are quite slow as compared to in-memory operations, it is likely that the thread will even be blocked, which means that the `addTask` operation often happens in parallel with the first class of necessary delay, waiting for a thread to finish with an old task.

2.5.2 Experimental Testing of System Responsiveness

We would like to be able to test the responsiveness under different conditions and different system design choices. In some cases, the advantages of a design change are obvious, the best example being the move from a database that stored individual objects to one that stored pre-compressed blobs (which will be described

further in chapter 3). The improvement in performance was easily visible to the naked eye. In other cases the advantages are less clear.

We have implemented a testing procedure that works as follows. The client is first run in a special mode, called *gui-trace* mode, in which it records every mouse event, along with a timestamp, and stores this information in a *trace file*. Then the client can be run in *run-trace* mode, in which it reads the trace file and triggers the stored mouse inputs, at the appropriate time intervals. The idea is to record a user session, and then re-run that same session under a variety of conditions, comparing the performance.

Unfortunately, there is a problem with this approach. Because of unpredictable network speeds and unpredictable timing among the threads and between the client and server, along with the reordering and cancellation of requests, two runs of the same trace are not even guaranteed to include the same events, much less the same events in the same order. For example, in one trace, the client sends nine range requests to the server, and after the server has responded to four of them the user jumps to a new location, and the client sends a *new-view* command, which causes the server to ignore the last five requests. But in a second run of the trace, the client may be slower, and the *new-view* command doesn't get to the server until after all nine requests have been satisfied.

We could accept the fact that different runs will have different events, and perform enough runs of the experiment so that the computed average performance is statistically significant, but the problem runs deeper than that: we don't have a clear idea of what we want to measure! The goal is responsiveness. If we have the client send some requests to the server, and then measure how long we have to wait until the system is at rest again, then it might seem that shorter wait times are better, but we are not getting an idea of how the system performs while in flux, which is essential. What we really want is to have the *right* data being transferred at the *right* time, but it is not at all clear how to measure this.

In lieu of a rigorous, automated testing method, we have performed some very simple tests by hand that demonstrate the effects of request cancellation. The results are shown in table 2.1. The basic test is to click on a button to jump to a new location, and use a stopwatch to time how long it takes from clicking the button until all the data has been downloaded from the server. (The user interface provides a visual indicator for when no more data is being downloaded.) At least three trials were done for each test, and the average taken. The first five lines in the table show the results of this test for five different cities. Then we tried combinations of cities, as seen in the last four lines of the table. In this case each button is clicked in quick succession, and after the last button we wait until

	measured	sum	% saved
New York	2.8 sec	-	-
Boston	3.1 sec	-	-
Washington, DC	2.5 sec	-	-
San Francisco	2.3 sec	-	-
Philadelphia	3.3 sec	-	-
Bos., NY	5.14 sec	5.9 sec	12.7%
DC, Bos., NY	6.2 sec	8.4 sec	25.9%
SF, DC, Bos., NY	8.1 sec	10.7 sec	24.4%
Phil., SF, DC, Bos., NY	9.6 sec	14.0 sec	31.3%

Table 2.1: Times in seconds between clicking the first jump button and system coming to rest, as measured by human with a stopwatch.

all data has been downloaded. Timing starts when the first button is clicked. The first column shows the measured times. For the multiple-click tests, the second column shows the sum of the times for constituent cities, which is what we would expect if the each city's data were allowed to download to completion before retrieving the next city, and the last column shows the percent time that is saved by not waiting—presumably from the cancellation of requests.

There is also the issue of measuring the aspects of responsiveness that are unique to the client side. One important measurement would be the amount of time between a mouse event being delivered to the client by the underlying graphical system, and that event being handled by our client program. (We have no control over how fast the underlying graphical system is.) Recalling the discussion of interaction times from chapter 1, we want this time to always be less than 0.1 sec. Setting up such a test is a matter for future work. In the meantime, we have measured the time to render a single polygon, which we know from claim 1 is an important factor determining the speed of the initial response to user input. Figure 2.6 plots the rendering time for a polygon and all of its descendant polygons in milliseconds against the number of points on the borders of the polygons rendered. While performance certainly drops for very large polygons, it seems from this limited testing that once a polygon is below about 1000 points it doesn't do much good to make it lower (by doing less merging—see section 3.3), and it would do harm if more polygons needed to be rendered.

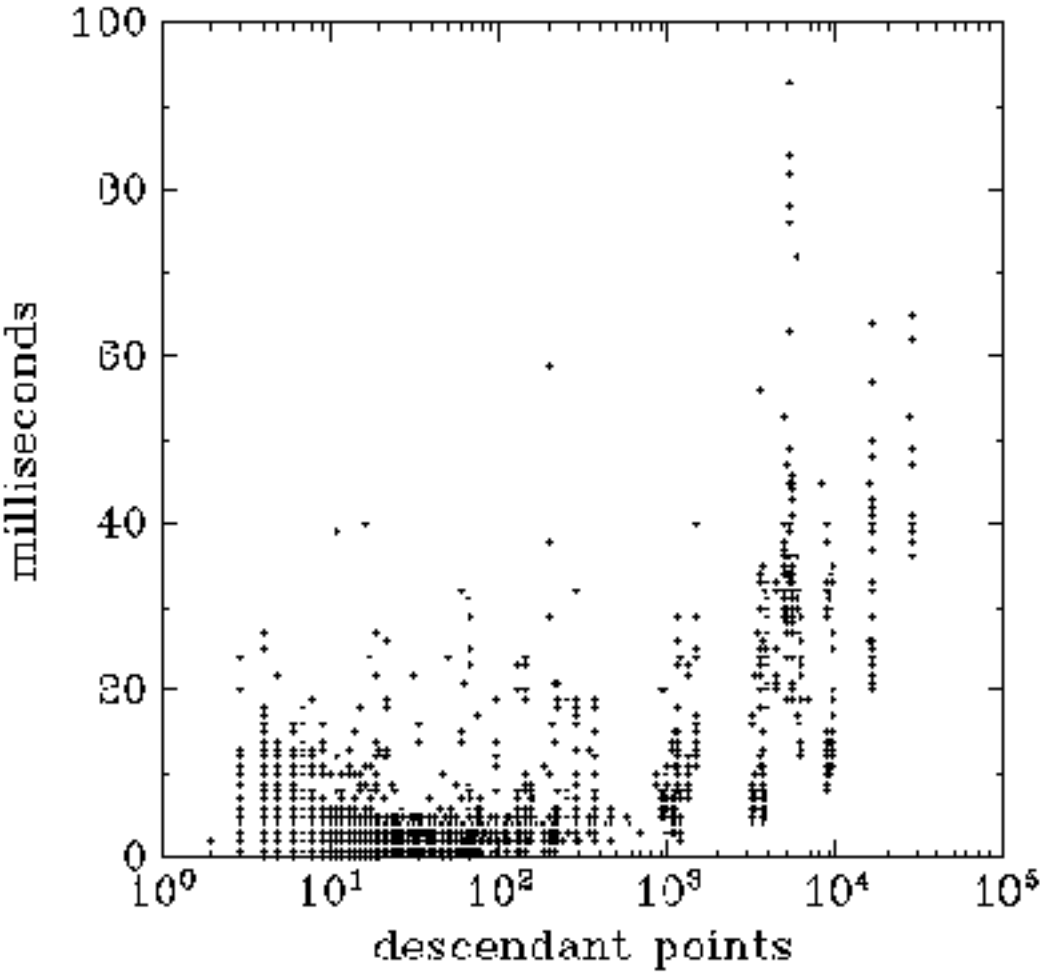

Figure 2.6: Polygon rendering time vs. number of points in polygon and descendants

2.6 Cache Consistency

As the client makes requests for data, and the server provides the data, the cache and cache copy will grow in size. So long as this is all that happens (and so long as neither the client nor the server runs out of memory), it is fairly easy to see that the system functions properly: every blob that is requested first has its id stored in the cache copy, then gets sent across the thinwire to the client, and then is stored in the cache and displayed on the screen. The order in which blobs are stored in the cache copy may not be the same as the order in which they are stored in the cache, but that is fine—every blob will eventually get to the client cache. (Actually, that is assuming the number of requests is finite. Without that assumption, a blob might get stuck on the Response Sender or Unpacker priority queue forever, while an infinite number of higher priority blobs pass it by.) But in order to not run out of memory, data must eventually be removed from the cache and its copy, and this is where we must be careful so as to make sure the system performs correctly. In this section we first give some examples of how things can go wrong, then give a correct solution, and finally prove the correctness of that solution.

2.6.1 Some Flawed Approaches

In the following, let b represent a blob, let C represent the set of blobs in the cache, let C' represent the set of blobs in the cache copy, and let S represent the set of blobs in the database that overlap the current screen—i.e., those that have data that should be shown on the screen. When the system is at rest, we should have $C = C'$ and $S \subseteq C$

One naive approach is to have the client decide which blobs to discard, drop them from the cache, and then send a list of blob ids to the server so the server can drop them from the cache copy. Unfortunately, this introduces a consistency problem. Consider the following sequence of events:

1. Start: $b \in C$, $b \in C'$, and $b \notin S$.

2. The Unpacker removes blob b from the cache. Now $b \notin C$, $b \in C'$, and $b \notin S$.

3. The user moves the current view to area R, which includes b. Now $b \notin C$, $b \in C'$, and $b \in S$.

4. The Searcher asks for objects in R from the cache, to send to the Renderer, but b is not found because it was just removed.

5. The Request Sender sends range request R to the server, but the server does not send b back, because it is in the cache copy.

6. The Unpacker sends a "remove b" command to the Request Sender, which forwards it to the server, and the server removes b from the cache copy. Now $b \notin C$, $b \notin C'$, and $b \in S$.

This is an inconsistency, even though client and server agree on the cache contents, because if there are no more requests the system will come to rest, and we do not have $S \subseteq C$. The client displays any new data in R that subsequently arrive from the server, but since this does not include objects from b, the objects in b will never get displayed. If another request for an area R' that contains b is subsequently made, then b will be sent, but there is no guarantee this later request will happen, and certainly not before the user becomes confused.

Another consistency problem to watch out for happens if, instead of having the client send a list of blob ids to the server for pruning, the server maintains in the cache copy a replica of the data structure in the cache, and then the client and server execute the same discard protocol on the replicated data structure. So the client executes the discard protocol on the cache, and then sends a message to the server instructing it to execute the protocol on the cache copy. Now consider this sequence of events:

1. Start: $b \notin C$ and $b \notin C'$.

2. The client requests from the server area R, which includes object b.

3. The client prunes area R' from the cache, which would cover b, except that the client hasn't downloaded b yet. The client then sends a prune R' message to the server.

4. The Data Provider puts b into the cache copy, and passes it on to the Response Sender. Now $b \notin C$ and $b \in C'$.

5. The Data Provider receives the prune message and removes b from the cache copy. Now $b \notin C$ and $b \notin C'$.

6. The client receives b and stores it in the cache. Now $b \in C$ and $b \notin C'$.

At this point the system can come to rest while the client has blob b, but the server thinks it doesn't. The next time the client requests an area that includes b, a duplicate will be sent.

These two problems can be solved by having the *server* discard objects first. The client is in a better position to decide when the pruning needs to happen, and what regions to discard, so it still makes those decisions first, but it delays doing the actual cache pruning until after it has notified the server of its decisions and the server has done the pruning. However, this protocol is not quite correct because or task prioritization. If the prune has a high priority, the following problem arises when the prune command "passes by" a blob on the way to the client:

1. Start: $b \notin C$ and $b \notin C'$.

2. The Data Provider adds blob b to the cache copy, and puts it onto the Response Sender queue. Now $b \notin C$ and $b \in C'$.

3. The Data Provider prunes region R, which includes b, from the cache copy, and puts the *prune* command onto the Response Sender queue. Now $b \notin C$ and $b \notin C'$.

4. The Response Sender processes the prune command first, because it has high priority, sending it to the client, where blobs in R are pruned from the cache. This would include b, but the client hasn't received it yet.

5. The Response Sender sends b to the client, where it is added to the cache. Now $b \in C$ and $b \notin C'$.

At this point the system can come to rest while the client has blob b, but the server thinks it doesn't.

If the prune has low priority, we have an analogous situation, where the prune is passed by a blob:

1. Start: $b \notin C$ and $b \notin C'$.

2. The Data Provider prunes region R from the cache copy, and sends the prune command on to the Response Sender. Region R would include b, if it were in C'.

3. The Data Provider adds b to the cache copy, and sends it on to the Response Sender. Now $b \notin C$ and $b \in C'$.

4. The Response Sender handles b before the prune, because it is higher priority, and sends it to the client, where it is added to the cache. Now $b \in C$ and $b \in C'$.

5. The Response Sender sends the prune command to the client, and the client removes b from the cache. Now $b \notin C$ and $b \in C'$.

Now if the system comes to rest, the server thinks the client has b, but it doesn't.

2.6.2 A Correct Approach

So what we really want is that the prune command will neither pass any blob nor be passed by any blob on the way from the Data Provider to the Unpacker. This is precisely the functionality provided by the flushAndPrune command. So here, finally, is the working protocol:

1. The client chooses a region R for pruning, and sends a prune(R) command to the server.

2. The Data Provider removes all blobs in R from the cache copy.

3. The Data Provider calls flushAndPrune(R) on the Response Sender queue.

4. The Response Sender sends all blobs already on its queue to the client, and then sends the prune(R) command to the client, before sending any new blobs that come after the prune(R) command.

5. When receiving the prune(R) command, the Response Receiver executes flushAndPrune(R) on the Unpacker queue.

6. The Unpacker puts all blobs that came before the prune(R) command into the cache, then removes all blobs in R from the cache.

2.6.3 Proof of Correctness

Before proving the correctness of this protocol, we must define what it means to be "correct". Informally, what we really want to say is that if the server should send a blob to the client, then it does, and if it shouldn't, it doesn't. But because the system is usually in flux, with blobs and prune commands in transition, it's not clear how to formalize this. We can't say that the contents of the cache and its copy are always the same, and we can't say that the same blobs get put

in and taken out in the same order. Instead, what we show below is that (1) for every request that arrives at the Data Provider without being canceled, all blobs that intersect the request range will at some time in the future all be in the cache together at the same time, and (2) the Unpacker never receives a blob that duplicates one already in the cache. These two results are represented by theorems 1 and 2 below.

First, we make some definitions. Let prune_i be the i-th prune command initiated by the Unpacker, and let $\text{prune}_i(x)$ represent the action of removing blob x from either the cache or the cache copy as part of the prune_i operation. (It will be clear from the context whether it is cache or cache copy.) Let $\text{add}_i(x)$ be the i-th time that x is added to the cache copy, the associated message for this event that is then passed to the client, and the action of adding x to the cache that is precipitated by that message. Let op_i refer to either prune_i or add_i. Before getting to the theorems, we establish some intermediate results in the following lemmas.

Lemma 1 *Every* op_i *that occurs at the Data Provider also occurs (later) at the Unpacker, assuming the number of user requests is finite.*

Proof: Whenever the Data Provider executes a prune_i or add_i operation, it puts the associated command onto the Response Sender queue. Since there are no clear commands executed on either the Response Sender queue or the Unpacker queue, the command will not be deleted before arriving at the Unpacker. Since the number of user requests is finite, the command cannot be infinitely delayed on a priority queue by being passed up repeatedly by more important tasks. Therefore, it will eventually get to the Unpacker. □

Lemma 2 *Every* op_i *that occurs at the Unpacker also occurred (earlier) at the Data Provider.*

Proof: The Unpacker only executes add and prune operations that come from the Response Receiver, the Response Receiver only sends commands that it receives from the Response Sender, and the Response Sender only sends commands that it receives from the Data Provider. (Note that the Unpacker *initiating* a prune command is not the same as *executing* the prune command. Initiating the prune command does not involve changing the contents of the cache.) □

Lemma 3 *At the Data Provider,* prune_i *precedes* prune_j*, for* $i < j$

Proof: At every priority queue, the prune command is placed onto its global queue, which is an ordinary first-in first-out queue. Since prune_i will be first-in to the Request Sender queue, it will also be first-out to the Request Sender itself, first across the thinwire to the Request Taker, first-in to the Data Provider queue, and first-out to the Data Provider itself. □

Lemma 4 prune$_i$ *precedes* op$_j$ *at the Data Provider iff* prune$_i$ *precedes* op$_j$ *at the Unpacker.*

Proof: First, note that lemmas 1 and 2 guarantee that the set of operations at the two locations are the same, so the lemma statement is well defined. The lemma then follows from the definition of the flushAndPrune operation. When the Data Provider processes the prune$_i$ command, it executes the flushAndPrune operation on the Response Sender queue, which means that a task op$_j$ on the Response Sender queue is processed by the Response Sender before prune$_i$ iff op$_j$ preceded prune$_i$ at the Data Provider. The Response Sender's behavior is to simply send messages across the thinwire to the Response Receiver, therefore the Response Receiver will in turn receive op$_j$ before prune$_i$ iff op$_j$ preceded prune$_i$ at the Data Provider. But for every task that the Response Receiver gets, it either puts it onto the Unpacker queue, or executes flushAndPrune on the Unpacker queue; therefore, when the Response Receiver processes prune$_i$, it will execute the flushAndPrune operation on the Unpacker queue, which means that the Unpacker also will receive op$_j$ before prune$_i$ iff op$_j$ preceded prune$_i$ at the Data Provider. □

Lemma 5 *At the Unpacker,* prune$_i$ *precedes* prune$_j$*, for* $i < j$

Proof: This follows from lemmas 3 and 4. □

Let C_i be the set of blobs that the Data Provider adds to the cache copy before prune$_i$, and after prune$_{i-1}$ if $i > 1$; let C'_i be the set of blobs that the Unpacker adds to the cache before prune$_i$, and after prune$_{i-1}$ if $i > 1$; let P_i be the set of blobs that the Data Provider removes from the cache copy as part of prune$_i$; let P'_i be the set of blobs that the Unpacker removes from the cache as part of the prune$_i$ operation; let S_i be the set of blobs in the cache copy immediately after prune$_i$, or the empty set if $i = 0$; and let S'_i be the set of blobs in the cache immediately after prune$_i$, or the empty set if $i = 0$. Then we have the following two lemmas.

Lemma 6 *For* $i \geq 1$, $C_i = C'_i$ *and* $P_i = P'_i$.

Proof: Lemma 3 implies that the sets C_i are well defined. Lemma 5 implies that the sets C'_i are well defined. $C_i = C'_i$ then follows from lemma 4. $P_i = P'_i$ can then be proved by induction. Since $C_1 = C'_1$, the cache and the cache copy are in the same state when the first prune operation is performed on each. Since the prune operation performs the same algorithm on each, and they are both in the same state, the same blobs will be removed, and $P_1 = P'_1$. Now suppose that $P_i = P'_i$ for every $i < j$. Since we also know that for all i, $C_i = C'_i$, the same argument shows that $P_j = P'_j$. □

Lemma 7 *For $i \geq 1$, $S_i = S'_i$.*

Proof: By induction. We know that $S_0 = S'_0 = \emptyset$. Suppose $S_i = S'_i$. Since $C_{i+1} = C'_{i+1}$ and $P_{i+1} = P'_{i+1}$ by lemma 6, it must be that $S_{i+1} = S'_{i+1}$. □

The following two theorems establish the correctness of the cache consistency protocols. What they are essentially saying, in less formal language, is that if the server should send a blob, it does, and if it shouldn't, it doesn't.

Theorem 1 *Let $B = \{b_1, b_2, ..., b_n\}$ be the set of blobs in the database that intersect the range R for a given range request. If the user makes only a finite number of requests, then for every range request R that arrives at the Data Provider, some time later there will be a time when all $b \in B$ will be in the cache, and that time will be before the next prune, if there is one.*

Proof: Consider any arbitrary $b \in B$. At the time that R arrives to the Data Provider, b is either in the cache copy, or not. Case 1: b is not in the cache copy. In this case, add(b) will be executed on the cache copy, and by lemma 1, add(b) will later be executed by the Unpacker. By lemma 4, that will happen before the next prune, if there is one. Case 2: b is in the cache copy. Here we have two sub-cases, depending on whether there has been a prune operation since add(b) was performed last by the Data Provider. Case 2.1: There has been a prune. Let prune$_i$ be the last prune before R arrived at the Data Provider. Since clearly $b \in S_i$, lemma 7 guarantees that it will be in the cache after prune$_i$, and will be there at least until the next prune, if there is one. Case 2.2: There has not been a prune. Then, by lemma 4, add(b) will be performed by the Unpacker, before the next prune, if there is one. Since we have considered every case, every $b \in B$ will be together in the cache, and before the next prune, if there is one. □

Theorem 2 *At the point that the Unpacker receives* add(b), *b is not in the cache.*

Proof: By lemma 2, the Data Provider previously executed the add(b) command, and by lemma 4 it must have been after prune$_i$, the most recent prune command at the Unpacker. Since b cannot have been in the cache copy before the Data Provider executed the add(b) command, b was not in S_i, and by lemma 7, b was not in S'_i. Since there can be only one add(b) command between any two prunes, b cannot have been added after prune$_i$ and before this add(b), so b was not in the cache when the Unpacker received the add(b) command. □

2.7 Discard Policy

It remains to decide what the pruning algorithm will be. In other words, which blobs will be removed from the cache and cache copy in any given prune. Some

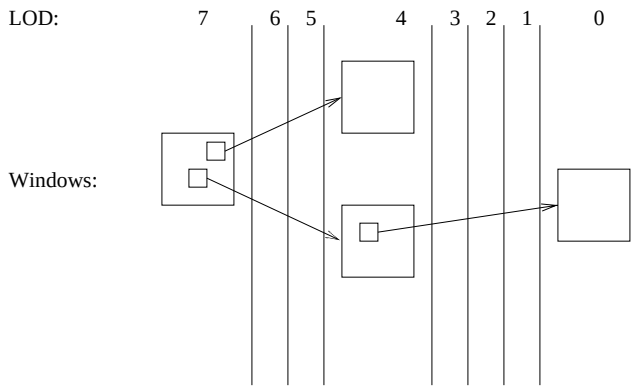

Figure 2.7: Sample LOD to Window Relationship

obvious possibilities are least recently used (LRU); first-in first-out (FIFO); and a strategy based on geography, in which data elements that lie in a discard range, or outside a *preserve range*, or outside multiple preserve ranges, are pruned.

The problem with FIFO is that if users pan around the map and then return to where they began, the data they are currently viewing might be deleted. One problem with LRU is that only the client knows the usage history, not the server, making it difficult for the client and server to implement the same algorithm. Probably the client would have to enumerate a list of the objects that it wants discarded and send those to the server, a performance hit that we would like to avoid. Also, it is not clear what the definition of "recently used" would be. If some objects are currently shown in a window, but those objects have not been searched for recently, and that window has not had the input focus recently, then are those objects "recently used"?

So a geographical approach seems best. (Matos, et al. [55], and Chim, et al. [15], come to the same conclusion.) We define at least one preserve range for each level of detail (LOD) based on the windows that are currently open. Figure 2.7 demonstrates a sample session in which four windows are open: the root window, at LOD 7, two windows at LOD 4, and one at LOD 0. If a level has windows open, we take the range currently showing in each window and expand it by a factor of two in each direction to produce the preserve ranges for that level. If there are no windows currently showing at a given level, then we use windows from nearby LODs as the starting point. There are three possibilities for the current level: (1) the only windows are at lower levels, (2) there are windows at higher and lower levels, and (3) the only windows are at higher levels.

For situation (1), we look for the nearest lower level for which there are

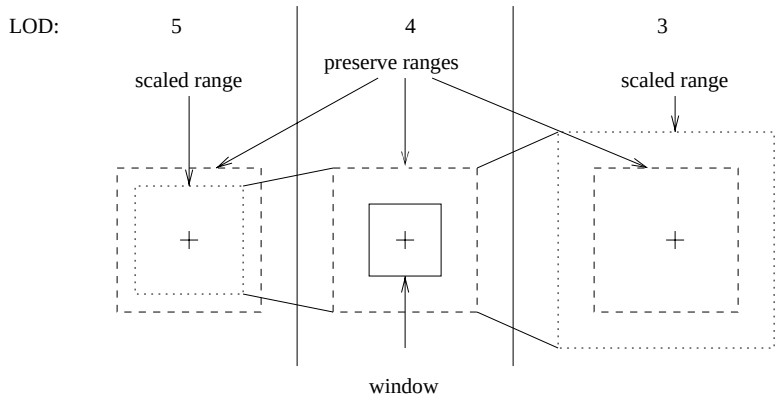

Figure 2.8: Sample Preserve Ranges

windows, and start with the ranges being shown in those windows. Similarly, for situation (3) we look for the nearest higher level that has windows, and use those ranges. For situation (2), we start by looking for the nearest level in either direction that has windows, and use the ranges for those windows. If there is a tie, then we use the windows from the lower level. This gives the center points of the preserve ranges that we are creating, but we still need to compute the range sizes. We start with the size of the range for the window that was used to determine the center point, and scale that range up to the "middle" range for the current level. The resolutions go up exponentially, so to define the "middle" scale for a level of detail we use the following:

$$S = S_{\min} \times e^{0.5 \log(S_{\max}/S_{\min})}$$

where $S_{\min}$ and $S_{\max}$ are the minimum and maximum scales, respectively, for the LOD. So if we are looking for a preserve range for LOD i, suppose the nearest window is w, in LOD j, with scale S_w. Let R_w be the preserve range of w, and let S_i be the middle scale for LOD i. Then the preserve range for LOD i will be R_w scaled up (or down) from S_w to S_i. Figure 2.8 shows a sample for one of the windows in LOD 4 from figure 2.7.

2.8 Searching the Cache

In this section we look at the data structure that is used to store data in the cache, and also in the cache copy.

2.8.1 Requirements

The data cache structure needs to support three operations:

1. find all data elements that intersect a given range,
2. add a new data element, and
3. remove all elements that *do not* intersect a given range.

The data structure will also be subject to concurrent accesses from different threads, but there are restrictions in the system architecture that limit the amount of concurrency that we need to worry about. On the server, only one thread, the Data Provider, reads and writes the cache, so there is no concurrency problem. On the client, the Unpacker writes the cache (both insert and delete), and two threads read it, the Searcher and the Event Handler. (Recall that the data flow transition from the cache directly to the Event Handler is not shown in figure 2.1.) Furthermore, since deletion is relatively rare we are willing to just lock the whole tree for deletion. Therefore, we only need to worry about one adder operating on the tree concurrently with multiple searchers.

We do not need full serializability. Consider the following sequence of events, in which it appears to the Searcher that geographic feature A is inserted before B, but for the Event Handler it appears that B is inserted before A:

1. The Searcher iterates partway into the cache.
2. The Unpacker adds geographic feature A to the portion of the cache that the Searcher has already scanned.
3. The Event Handler iterates completely through the cache, returning A.
4. The Unpacker adds geographic feature B to the cache.
5. The Searcher completes its iteration, returning B, but not A.

To see why this is not a problem, notice that the purpose of the Searcher is to send objects to the Renderer. But when the Unpacker unpacks an object, it both puts that object into the cache and sends it to the Renderer, which means that A will be sent to the Renderer, just as if the Searcher had found it. So the overall effect is equivalent to a history where the Event Handler does its search after A is added to the cache, and the Searcher does its search after both objects have been added to the cache—a serial sequence of events. More generally, any object

that is added to the cache concurrently to a search by the Searcher will seem to have been found by the Searcher, and so if there is also a concurrent search by the Event Handler, the system as a whole will always see an equivalent serial history in which the Searcher's transaction was after the Event Handler's.

The requirements, therefore, are fairly relaxed. Mainly we just don't want the structure to become inconsistent, or to have objects disappear temporarily (or permanently). Ideally, we would like each thread to never lock more than one tree node at a time.

2.8.2 The R-tree

Many database systems, including PostgreSQL, use the R-tree [37] for geometric indexes. It shares many of the features of the B-tree that make it suitable for database applications, including the ability to match node size to disk page size. Unlike the B-tree, however, the R-tree stores 2-D geometric data, which does not have an ordering. Each node is associated with an MBR, and the MBRs for the children of a node can overlap. Searching is done using the MBRs, which means that in a search, multiple children of a node might need to be searched. Thus, the R-tree cannot guarantee logarithmic searching, but in practice the performance has been found to be good.

In both the B-tree and the R-tree an update at a leaf may cause further updates all the way up to the root, which makes it difficult to provide a high level of concurrency. The B-tree "with links" data structure [7] mostly solves this problem, by allowing the insertion or deletion algorithm to only lock one node at a time, but it depends on a linear ordering of the data elements, which does not exist with R-trees. (For full serializability, a searcher needs to hold read locks on all leaves that it reads until the end of the transaction. But as mentioned above, we don't need full serializability.) Kornacker and Banks [48] found a way to get around this problem, but their solution still requires *lock-coupling*: a child node must stay write-locked until a write-lock is obtained on its parent. For database applications in which I/O operations are the primary concern, the lock-coupling "should make little difference to the achievable degree of concurrency", because the parent node is likely to be in main memory, and so "no locks are held during I/O operations".

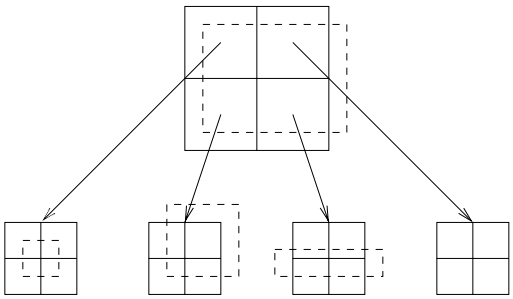

Figure 2.9: QR-tree Node with Children

2.8.3 The QR-tree

Since our cache is all in memory, we do not need the aspects of the R-tree that make it suitable for disk I/O, and we want to avoid lock coupling. We have used circumstances unique to our system, including knowledge of limited concurrency requirements and knowledge of the minimum bounding rectangle of the entire data set ahead of time, to design a custom data structure. We call it the QR-tree, since it incorporates some aspects of the R-tree and some aspects of the Quad Tree [25, 73, 74].

Each node in a QR-tree has associated with it a minimum bounding rectangle, or MBR, and a *region*, which is also an axis-parallel rectangle. The region is fixed at the time the node is constructed, whereas the MBR is updated every time a data element is added to the subtree. The region for the root of the tree is the MBR for the entire data set, which we are assuming is known ahead of time. (If all elements of the data set were added to the tree, the region and the MBR associated with the root node would be identical.) Each node has four children, and the regions of the children are the northeast, southeast, southwest, and northwest quadrants of the parent's region. Figure 2.9 shows a node and its children. The regions are represented by solid lines, and the MBRs by dashed lines. Notice that the parent's MBR is the bounding rectangle of each of the child MBRs, and that the northeast child is empty, even though the parent's MBR intersects with it. Also notice that sibling MBRs may overlap, but their regions do not. The purpose of the regions is to give some added intelligence to the node addition algorithm, so that overlap between a node's subtrees can be minimized.

First, consider the QR-tree search, add, and delete operations without any consideration to concurrency.

QR-tree Search

Searching a QR-tree for objects in a given range is the same as searching an R-tree: at an internal node, if the node's MBR intersects the search range, we search each of the node's children; at a leaf, simply compare the data element to the search range.

QR-tree Addition

When adding a data element to an internal node, we pick the child whose region contains the center of the MBR of the new data element, and add the data element to that child. If that child is empty, we create a new leaf. When adding to a leaf, we replace the leaf with an internal node, and add the original leaf data and the new data element into the newly created internal node. The MBR of a node is expanded to include the MBRs of any added data elements. This way, the

In the case of adding to a leaf, if the original leaf data and the new data element have MBRs that are centered on the same point, then we have an infinite recursion, where the original data element is added to child i of the newly created internal node, creating a new leaf, and the new data element is also added to child i, which is now a leaf, so the process repeats. To prevent this problem, we need to force an end to the recursion by forcing the elements into separate children. This means that one of them will be in a child whose region does not contain its MBR's center point. Our approach is to end the recursion when the node that the elements will be added to has a region that does not completely contain the MBR of either data element. That way, both data elements can be added to a node with a region that at least overlaps the element's MBR. But now we have a new problem: consider the case where we add a new data element to a leaf which has an element whose center point is not within the region. After converting the leaf to an internal node, when we go back to add the old and new data elements, if we use the standard method of choosing the child whose region contains the center of the data element's MBR, we will be stuck, because no child contains the center of the old data element's MBR; the parent doesn't even contain it. Therefore, in this special case we use a different criteria for choosing a child: pick the child for which the area of the bounding rectangle of the union of the region and the data element is smallest.

QR-tree Deletion

For this project, we want to delete all data elements outside a particular range, called the "preserve range". The protocol is fairly simple; this executes at each node, starting with the root:

- if I am a leaf
  - if my MBR is intersects the preserve range, do nothing;
  - else set myself to the empty tree;
- else (I am an internal node)
  - if my MBR is within the preserve range, do nothing;
  - else if my MBR does not overlap the preserve range, set myself to the empty tree
  - else
    - for each of my children
      - if the child's MBR is within the preserve range, do nothing;
      - else if the child's MBR does not intersect the preserve range, remove it;
      - else recurse on the child;
    - recompute my MBR as the minimum bounding rectangle of each remaining child's MBR;

If we want to return a list of the objects that have been deleted, we only need to iterate over each deleted subtree.

QR-tree Experiments

We have done some crude experimentation with the QR-tree, comparing it to a similar class of trees in which nodes have only MBRs, and not regions. Searching on these other trees is the same as with the QR-tree, but for adding a new element to an internal node, we must now decide which child gets the new element using only the MBRs of the children and of the new element. Frequently the new element will only partially overlap the MBRs of any existing child, so we have a choice between adding the new element to a subtree that is empty (essentially, creating a new child), or adding to a subtree that already has elements. We tried three different add algorithms, giving three variants. The first variant favors

creating new children, and thus tends to produce a short, wide tree; the second variant favors adding to subtrees that already have elements, and thus tends to produce tall, narrow trees; and the third variant is in the middle. Note that since there is overlap among a node's subtrees, short and wide is not *a priori* better. A number of elements were inserted into each tree, and then a number of queries were performed, and the number of nodes traversed in each query was measured. Of course we expect that a query that returns more nodes in its result set will traverse more nodes in the search, so we divided the number of nodes traversed by the number of nodes returned. For number of nodes traversed per node returned, we found that the QR-tree averaged 4.2, the wide variant averaged 7.1, the middle variant 8.6, and the narrow variant 12.0. This seems to indicate that associating a region with each node does improve performance. More extensive testing is needed, and comparisons with the R-tree would also be interesting.

2.8.4 The QR-tree and Concurrent Access

We have a method by which adders and searchers never have to lock more than one node of a QR-tree at a time. Our solution uses two atomic operations, which operate on a single node of a QR-tree: `getDataIfLeaf()` and `setChildren(children)`. The `getDataIfLeaf()` method returns the stored data element if the node is a leaf, and returns a null pointer if the node is not a leaf. The `setChildren(children)` method sets the node's children pointer to the given children pointer (a pointer to an array of node pointers), and sets a flag indicating that this node is now an internal node. Both of these operations are extremely fast, so the overhead in providing synchronization and the amount of time that each thread spends in a blocked state on a QR-tree are quite small. Given these two atomic operations, we can now describe the search and add algorithms.

The search method should return a subset of the data elements in the tree, containing elements that match a given criteria (typically, intersecting a search range). It starts by finding the first leaf in a depth first search. Now, we would like to just inspect the data element at that leaf, and if it meets the criteria, add it to the return list. But there is a problem: in between the time that we discover the leaf and the time that we retrieve its data element, an adder may have turned the node into an internal node. So we execute the `getDataIfLeaf()` method. If it returns a data element, the element is checked and possibly returned; if it returns null, we continue with the depth first search until we find the next leaf. In this second case, the first subtree searched will be the one rooted at this internal

node that was previously a leaf. As long as there are not an infinite number of add operations, the search will eventually find the data element again, in its new leaf, and add it to the return list. After finding a data element at leaf node n, checking it against the search criteria and either returning it or not, when we start searching for the next leaf, we start from n *as if* it is still a leaf, even if it is not. In other words, if it has in the meantime been turned into an internal node by an adder, we will not search the newly created subtree. This is necessary for correctness, as will be explained below.

The add procedure is the same at every node. First, the node's MBR is expanded (if necessary) to cover the bounding rectangle of the new data element. Then we have three possibilities:

1. *The node is empty.*

   Set its data pointer to point to the new data element, and set a flag to indicate that the node is now a leaf.

2. *The node is an internal node.*

   First compute, according to the four sub-regions of this node, which child should take the new data element. If that child pointer is currently null, simply set it to point to a new leaf. Otherwise recursively add the new data element to that child node.

3. *The node is a leaf.*

   In the normal case, we want to convert the node to an internal node, and add the new data element, as well as the original leaf element, back into that newly created internal node. But because of concurrency considerations, we have to be careful how it is done. First, create `children`, an array of child pointers, and set the appropriate element to point to a new leaf node for the original data. Then call `setChildren(children)` on the original leaf node, to atomically turn it into an internal node with the original data element in a child node. Finally, recursively add the new data element to this node, which is now an internal node.

   In the case when we need to stop the recursive descent, as described in the part on addition in section 2.8.3, the difference is that instead of waiting until after `setChildren()` is called to add the new data element, we choose an index in `children` for the new data element right after choosing for the original data element, with the restriction that they cannot go into the same child. Then a new leaf node is also created for the new data element, two

child pointers in `children` are set, and finally the `setChildren()` method is called. Now both data elements are in the tree, so we do not need another recursive add.

Notice that, right up until the moment that the adder calls the atomic operation `setChildren()`, the adder has made no change to the tree that would be visible to any searcher, other than expanding the MBRs. After the `setChildren()` operation, the adder makes no other changes to the tree.

There are a couple of interesting situations. After an adder has updated the MBR of an internal node, but hasn't added the new data element to the subtree yet, a searcher that would return the new data element might traverse the subtree, even if no other data elements intersect its search range, but it will not find the new data element. The effect will be as if the search completed before the add happened. The only disadvantage is that the searcher might do some extra work in traversing a subtree unnecessarily.

After a searcher has retrieved the data element from a leaf, but before the next leaf in the traversal has been found, an adder may turn the leaf into an internal node, pushing the original leaf data down into a new subtree. If the search traversal continued down into the new subtree, the original data element would be returned a second time, which is clearly a mistake. This is why, as mentioned above, the searcher will continue the traversal *as if* this node is still a leaf, ignoring the children it now has. Then it's just as if the add operation happened after the search.

Of course the QR-tree is not general purpose, because its correctness is limited to the specific situations discussed above. Nonetheless, for our purposes, it has some advantages over the R-tree with links:

1. There is no lock-coupling.
2. A searcher almost never locks an internal node. (The only exception is when the searcher finds a leaf, but an adder turns the leaf into an internal node before the searcher gets the lock on it.)
3. An adder never needs to lock more than one node.
4. It is simpler to implement.

2.8.5 Tree of Trees

Because the data is packaged in blobs in the database and in the server cache copy, we will want to prune the cache in blob chunks, rather than in chunks of

individual objects. This requires a slight change in the discard policy: instead of deleting objects that do not intersect any of the preserve ranges, we must delete all objects *that come from a blob* that does not intersect any of the preserve ranges. Searching, however, must be done at the object level, so we need a structure that makes access easy for both blob granularity and object granularity. Our solution is to use a tree of trees. The main tree is a regular QR-tree with each leaf node associated with a blob. The data of each leaf node is then a QR-tree of the lines (or polygons or landmarks) that came in that blob.

For adding, when the client starts unpacking a blob, it starts three new QR-trees, for landmarks, lines and polygons. Each of these (initially empty) trees is added to the main tree for its data type. While that blob is active, each object unpacked is added to the appropriate inner QR-tree, and thus automatically to the appropriate main tree.

For searching, we simply need to change the search method on the main QR-tree so that when it gets to a leaf, instead of simply returning that leaf, it implements the search method on the (ordinary) QR-tree that is stored at that leaf.

Chapter 3

Preprocessing the Dataset

The most time-consuming aspects of this project have been the relatively mundane ones that involve preparing the dataset: interpreting the raw data format, translating the data into a more convenient format for the runtime system, unifying the county-based multi-file data into a single dataset, finding and handling various anomalies and mistakes in the raw data, and generalizing the data into multiple levels of detail. This is consistent with other researcher's experiences. In Funkhouser's architectural walkthrough project [33] it took almost as much time to construct the model, including converting 2.5D architectural models into a 3D representation suitable for rendering and creating multi-resolution models for various pieces of furniture, as it did to develop algorithms to visualize it. Kernighan and Van Wyk [45] gave detailed descriptions of the highly ad hoc heuristics they had to develop for converting AutoCAD DXF files into a more convenient geometrical format.

3.1 Related Work

In the architectural walkthrough system of Funkhouser, et al. [32, 29, 33], data is stored in segments, where each segment is "a variable-sized contiguous group of bytes in a display database file that can be read and released as a unit" [32]. They do not use a general purpose DBMS. I/O and memory allocation are for segments, not individual objects. Segments for all objects incident on a cell (e.g., a room) are stored contiguously in the file, and segments are grouped together using visibility information. Border objects used in more than one cell are duplicated, once for each cell. Within a cell, each object can be described at multiple levels of detail, and object descriptions are grouped by level-of-detail (LOD). Discrete

LODs are set during the preprocessing phase, and chosen dynamically at runtime. All LODs for a cell are in the same segment. They use a "one-level modeling hierarchy", which means that objects cannot reference other objects (i.e., no object containment hierarchy). A variant of the k-D tree is used to make the spatial subdivision to divide the data into cells, but no index is used at runtime for data access. Instead, there are data pointers from segment to segment based on relationships such as adjacent, incident, visible, etc., and runtime traversal is based on these.

In Schmalstieg's network-based walkthroughs [78, 77, 79, 43], "The database of the server is a flat collection of objects. Each object is composed of a geometry representation (trunk plus LODs), and a matrix defining the object's position and orientation." [77] As with Funkhouser's system, there is no index on the server; object pointers are traversed. Therefore neither of these systems are well suited to support sudden jumps to a distant viewport, as is necessary in our system.

Moving to GIS oriented projects, van Oosterom, et al. [94, 95, 96, 101, 100, 97], developed a sophisticated combination of data structures that have been implemented, most impressively, in a spatial data management system for the Netherlands Cadastre [99]. This is a database of more than 50 GB containing spatial and topographic data along with information about real estate parcel ownership and transactions, with histories. These new data structures and techniques allow all levels of detail for the geographic data to be stored in a single data structure, avoiding redundancy and its "related drawbacks: possible inconsistency and increased memory usage" [100], but still indexable by an R-Tree style approach. Two key components of map generalization, feature selection and line simplification, are done on-the-fly. This ambitious goal was achieved by the use of three new data structures: the BLG-tree, the Reactive-tree, and the GAP-tree. The BLG-tree (for "Binary Line Generalization") is used for on-the-fly polyline simplification. It encapsulates the simplification that would be obtained using the Douglas-Peucker algorithm [22], so that the simplification with any error bound can be quickly pulled out from the tree without re-executing the algorithm. The Reactive-tree provides two essential functions: spatial indexing and on-the-fly feature selection. It provides spatial indexing in a similar way to the R-tree. However, each object is assigned an importance, and less important objects are never stored at higher levels than more important ones. (Unlike the R-tree, leaves can occur at any level of the tree.) Selection is accomplished by only descending the tree far enough to get objects whose importance is above a particular threshold. Each polyline is represented in its leaf node by a BLG-tree. The GAP-tree (for "Generalized Area Partitioning") is an auxiliary structure

that is used to decide how to fill the "holes" that are left when an area feature is not selected. These data structures were implemented as extensions to Postgres. So in the end, all of the geographic data is stored in one table, with the Reactive-tree as the index, and every query must specify a range, an importance threshold, and a line simplification error bound.

The ViRGIS system [65, 63, 67, 87], is a virtual reality interface to a GIS, or a kind of GIS walkthrough. All data is stored in an extended ObjectStore OODBMS. The data includes terrain and texture in addition to geometric objects. The objects are indexed by an R-tree; how it handles multiple LODs is not explained. Terrain data is in one multiresolution quadtree. Texture is managed by a tiling scheme and indexed by a hashing structure, with different LODs in different databases.

3.2 The TIGER Data

The dataset we use is the TIGER data, a vector-based geographic description of the United States that is freely available from the US Census Bureau [92]. (TIGER stands for Topologically Integrated Geographic Encoding and Referencing.) A full description of the format can be found in the TIGER technical documentation [91]. Here we only consider those aspects that are important for this project.

3.2.1 Overview

The focal point of the TIGER data is the *complete chain*, which is really a polygonal line. In what follows, we refer to complete chains as simply "lines". Each line has two *endpoints* and zero or more *shape points* in between. In figure 3.1, lines L_4, L_5, and L_{10} have shape points, and the rest have only endpoints. The two endpoints of L_{10} have identical coordinates, so L_{10} forms a loop. The shape points always have degree 2, so when two or more lines meet at a point, that point must be an endpoint for each of the lines. An endpoint that is shared by multiple lines, or that is the endpoint at both ends of a single line, is duplicated in the dataset. Each line has a *feature code* and a *feature name*. The feature code is three characters long, and defines what type of feature it is—road, river, etc. For example, "A11" is the code for "primary road with limited access or interstate highway, unseparated". The feature name is up to 30 characters long, and gives a human-readable description, such as "Madison Ave" or "I-95". The

feature name may be empty, as is the case for many "non-visible" lines, such as municipal or political boundaries.

The lines connect so as to divide the plane into polygons. The entire surface area of the United States is covered by polygons, and polygons can overlap only if one polygon is completely contained within another. (In that case we can think of the outer polygon as having a hole.) In figure 3.1 there are four polygons, and polygon P_4 is completely contained within polygon P_3. A boolean flag for each polygon indicates whether the polygon maps to land or water. Most polylines serve as the common boundary of two polygons, but those that are on the outer boundary of the entire dataset serve as the boundary of only one polygon, and some polylines that lie within a polygon do not serve as any polygon boundary. In figure 3.1, line L_9 does not serve as a polygon boundary.

The TIGER data also delineates special or notable places called *landmarks*. The landmarks are divided into two classes: *point landmarks* and *area landmarks*. Figure 3.1 contains one point landmark, in polygon P_3. As with the lines, each landmark has a feature code and feature name, and the name may be empty. Each point feature is associated with an independent point that is not coincident with any endpoint or shape point. Each area landmark is associated with one or more polygons, but not all polygons are associated with a landmark. In our system, however, all polygons are treated equally; those that are not landmarks are assigned a generic feature code and an empty name. Since area landmark properties are incorporated into the polygons themselves, we do not need to refer explicitly to area landmarks, and so for simplicity we refer to point landmarks as simply "landmarks".

3.2.2 File Organization

The data comes in the form of more than 3000 "zip" archive files—one for each county or county level entity. (In Louisiana, the counties are called "parishes". In Virginia, there are many towns that are independent entities, not part of any county. These are treated like counties in the TIGER data, and we use the word "county" to mean any of these.) A zip archive contains as many as 17 files. Each file contains a distinct record type, and each record type is given a single character alphanumeric code. We are interested in only six of the record types.

Record type 1 (lines) There is one record of type 1 for every line. This record contains a unique id, feature code, feature name, and coordinates of the endpoints, along with many other fields that we do not use.

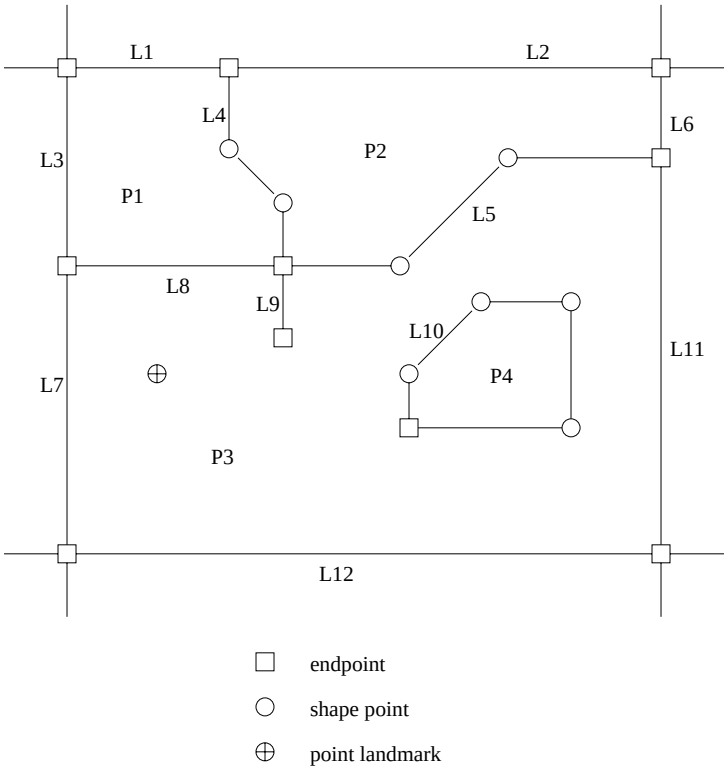

Figure 3.1: TIGER Data Diagram

Record type 2 (shape points) There can be zero or more type 2 records for each line. These contain the shape points, at most ten per record, along with the line id that matches the type 1 record and an index number. For example, if a line has 36 shape points, there will be 4 type 2 records with that line id, having index numbers 1, 2, 3, and 4. The first three will have ten shape points, and the fourth will have six shape points.

Record type 7 (landmarks) Each type 7 record describes a point or area landmark, including a landmark id, feature code, feature name, and latitude and longitude coordinates if it is a point landmark.

Record type 8 (polygon/landmark links) Type 8 records map area landmarks to polygons. Each record contains the landmark id and the polygon id.

Record type I (line/polygon links) Type I records link polylines to polygons. There is one record for each polyline. Each record contains the line id and the polygon ids for the right and left sides.

Record type S (polygon descriptions) A type S record for each polygon contains many fields describing the polygon. We only use one boolean flag, which tells whether or not the polygon represents a water feature (lake, river, etc.).

Lines that form a border between two counties will appear in both counties, but possibly with a different line id. Whether they have the same line id or not, the two representations for a county border line must have the same endpoints and the same shape points. (As will be described in section 3.2.3, we have found two cases where this requirement was violated, but these are errors in the dataset that should be corrected in later versions.) The records for county border lines do not indicate the identity of the neighboring county. Section 3.5.5 describes how we recover this information.

3.2.3 Anomalies

The normal case is that within each county, the set of lines partitions the plane into simple polygons, and the union of these polygons makes a simple polygon for the whole county. There are some anomalies, however, that make the preprocessing difficult. The images in this section are screenshots of our data selection tool, which is described in section 3.6.1.

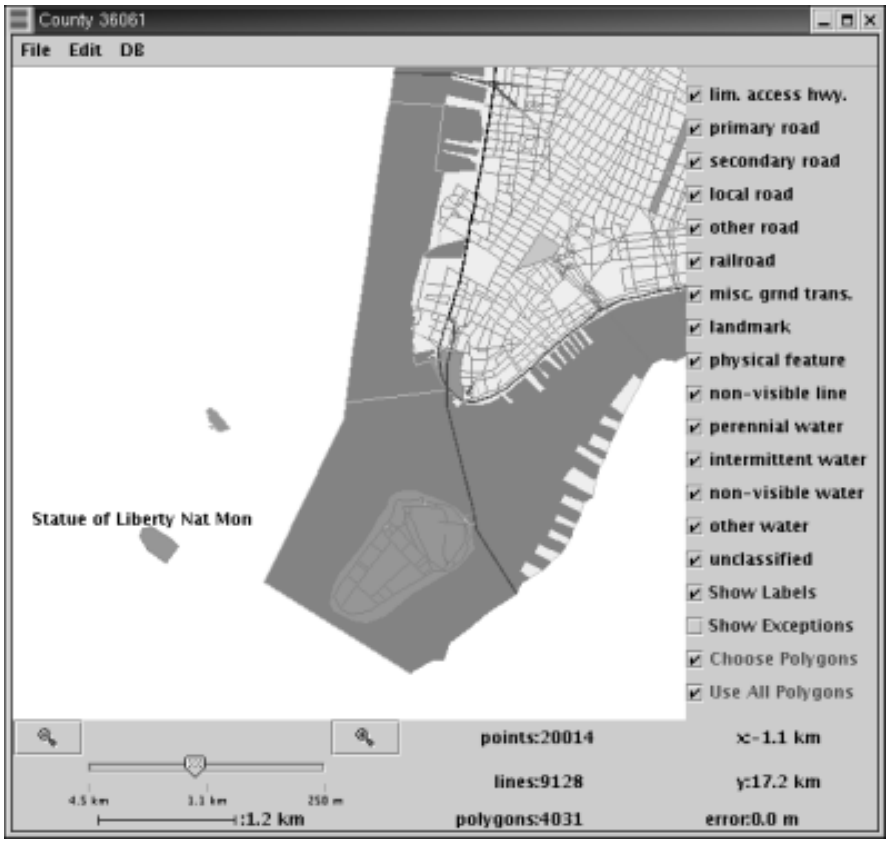

Figure 3.2: Statue of Liberty National Monument

Not all counties are connected, and the counties that are connected might have holes. For example, New York County, New York (Manhattan) has three pieces: the main island, the Statue of Liberty, and part of Ellis island. Both the Statue of Liberty and the New York part of Ellis Island are surrounded by land or water that is in Hudson County, New Jersey. Figure 3.2 shows the lower part of Manhattan, and the two islands, which are both part of the Statue of Liberty National Monument. In some cases, an entire county is surrounded by another county, or even surrounded by a single polygon within another county. The nesting can go deeper: in one case, the town of Manassas, Virginia is contained within Prince William County, Virginia, and a separate piece of Prince William County is in turn contained within Manassas.

Figures 3.3 to 3.7 cover this situation in Prince William County, which illustrates many of the anomalies we have to deal with. Figure 3.3 shows the part of Prince William County with the hole. The hole is broken up in the middle by three "islands", and in the upper right by a strip of land that goes across the hole. (The three islands look like one island in figure 3.3; figure 3.6 has a close up of the islands.) Figure 3.4 shows Manassas Park, which has two pieces that occupy the upper right portion of the hole in Prince William, straddling the strip of land that crosses the hole. Figure 3.5 shows Manassas, which occupies the rest of the hole in Prince William. Manassas itself has holes to accommodate the islands in Prince William, but they are difficult to see in figure 3.5. Figures 3.6 and 3.7 show closeups of the islands in Prince William and their holes in Manassas. Note that there are three islands, but at first glance figure 3.7 only appears to have one hole, which fits the island on the right. The reason has to do with the way that the data selection tool, from which these screenshots were taken, shows holes. It will show the hole as the background color, white, if the hole is occupied by a polygon that is not contained within another polygon, which we call a *top-level polygon*. If a hole is completely within a single polygon, then the hole is only shown as an outline. In figure 3.7, the outline for the two leftmost islands in figure 3.6 can be seen in the larger polygon to the left of the more visible hole.

This example clearly shows that there is no containment hierarchy for counties. Also note that we cannot determine that a county is contained inside another county by merely looking at its immediate surroundings: both Manassas and Manassas Park are adjacent to *two* other counties, which under a naive approach would not suggest that they are inside a hole.

Note: the Census Bureau releases new versions of the TIGER data fairly regularly, and to avoid topological errors and other inconsistencies when unifying the multi-county data it is important to use the same version for all the counties.

Figure 3.3: Prince William County, Virginia. Contains a hole, islands inside the hole, and a strip dividing the hole.

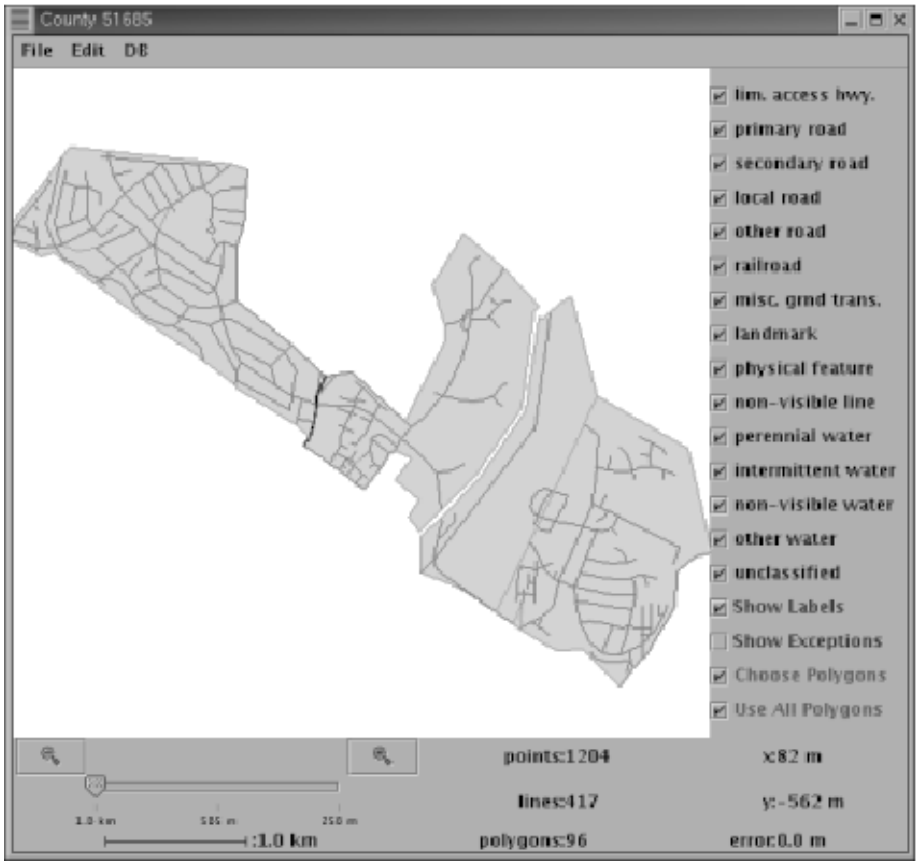

Figure 3.4: Manassas Park, Virginia. Fits inside the upper right portion of the hole in Prince William, straddling the strip across that hole.

Figure 3.5: Manassas, Virginia. Fits inside the hole in Prince William, and the hole in the middle of Manassas can just barely be seen.

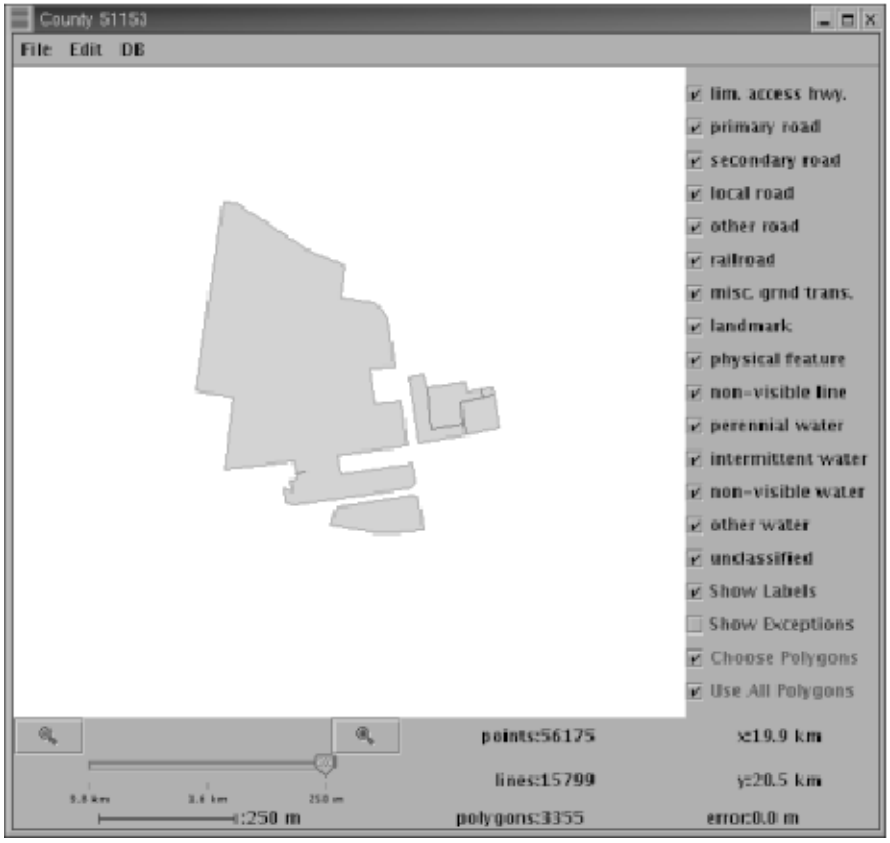

Figure 3.6: Closeup of the islands in the middle of Prince William County, Virginia.

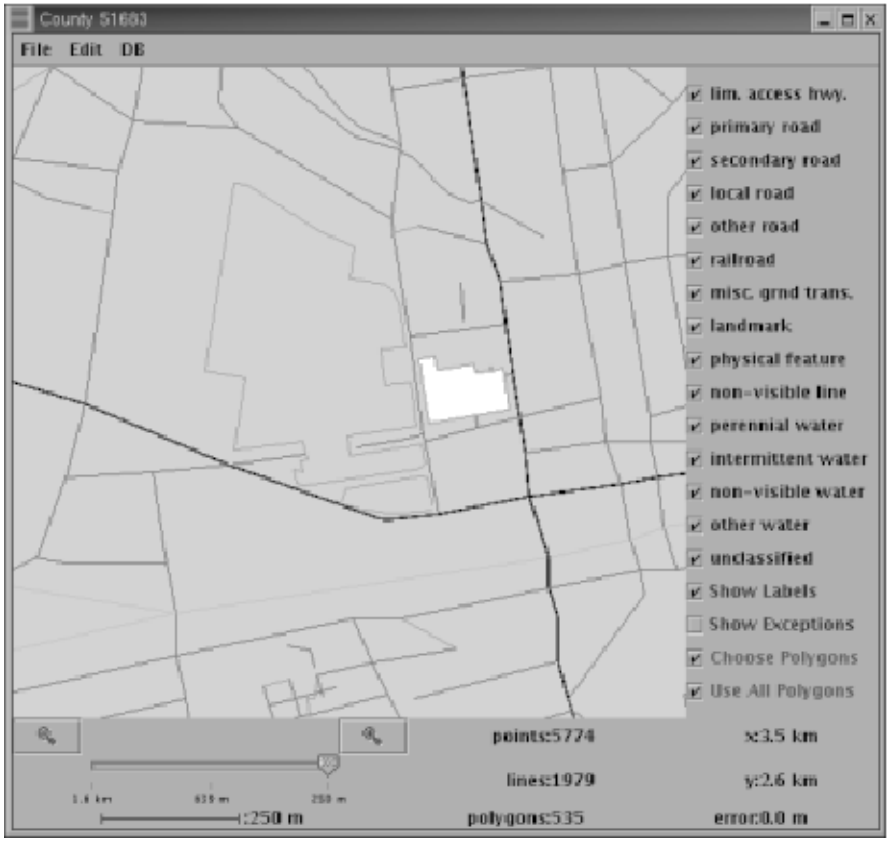

Figure 3.7: Closeup of Manassas, Virginia. The hole in the middle, and the two outlines next ot it, match the two islands in Prince William County.

For this project, we decided to standardize on the Redistricting Census 2000 TIGER/Line files, which is not the most recent version, but it is the "official" version that was submitted to the states for redistricting purposes. We have discovered two errors in this version that as of this writing were not documented on their "errata" page (though the Bureau has been has been notified of the errors). For the record, we list them here.

1. On the border between Clermont County, Ohio and Warren County, Ohio, the line with id 118457848 in Clermont is represented by two lines in Warren, with ids 159041118 and 159041119. Line 118457848 should be split to match the two lines in Warren.

2. On the border between Clay County, Missouri and Jackson County, Missouri, the single line 89768904 in Clay should be split twice to match the three lines 91515084, 91515090, and 91414091 in Jackson.

3.3 Merging the Raw Data

Because of the restriction that shape points can belong to only one polyline, the TIGER data is forced to have a large amount of duplication that we don't need for this project. In an urban setting like New York, for example, every rectangular Manhattan block must be a polygon, and each side of the blob must be a line. At a typical street corner, the point representing that corner will have to be listed in the data four times—as an endpoint of each of the four lines that are incident upon it. Each line in turn is referenced as the side of two polygons (though the line description itself is not duplicated). For this project we *merge* adjacent lines and polygons to produce a set with fewer but larger objects. This significantly reduces the size of the dataset. The left side of figure 3.8 shows the typical situation at a street corner. The four endpoints within the dashed rectangle all have the exact same coordinates, but they are shown separately for clarity. On the right side of the figure, the four lines have been merged down to two, reducing four endpoints to two shape points. When two polygons are merged, any lines that serve as a common boundary between them become internal to the new merged polygon.

Our merging method is to merge polygons first, and then merge lines based on the new set of polygons. In our simplest *unrestricted* approach, any two adjacent polygons in a county that have the same feature code and feature name are merged. Then, two lines that are incident on the same point and that have the

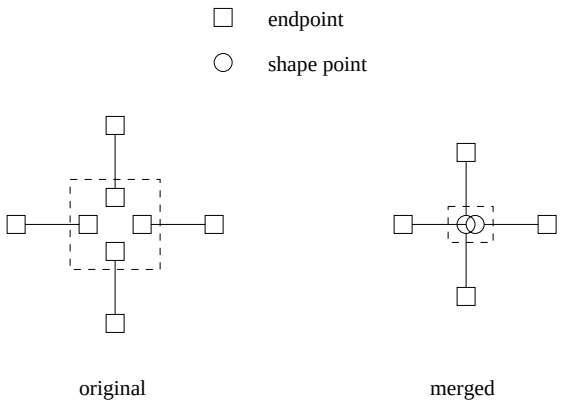

Figure 3.8: Before and After Merging

same two polygons on either side can be merged. In this case left and right don't matter. As an example, consider again figure 3.1. Suppose that polygons P_1 and P_2 have been merged into a new polygon, called P_5, and now we are testing to see if lines L_8 and L_5 can be merged. If the point where the two lines meet is the start point for both points, then L_8 has P_3 on the left side and P_5 on the right side; L_5 has P_3 on the right side and P_5 on the left side. Regardless of whether P_3 and P_5 have the same feature code and feature name, lines L_8 and L_5 can be merged. L_4 and L_9 cannot be merged, because L_4 has P_5 on either side, and L_9 has P_3 on either side.

Notice that after the merging step, no line can intersect more than one polygon, unless it is a common border between two polygons, and every polygon is either a top-level polygon, or the child of one other polygon. Since landmarks have zero extent, they trivially must intersect only one top-level polygon. Therefore the features can be partitioned according to their association with a particular top-level polygon, with the border lines being associated with two top-level polygons. This partition is used in the database schema, as described in section 3.4.2.

Also note that polygons that are top-level in the raw data may become children after merging. In fact, after the merging step, most polygons are children, although most of the surface area is covered by the larger top-level polygons.

Unrestricted merging tends to produce merged polygons that are quite large—in many cases the size of the entire county. Since at runtime we will be downloading information at the granularity of the merged top-level polygons, having very big polygons means that more data than is necessary will be downloaded;

counties can be much larger, in the projected viewing space, than the size of a typical window. So we want to strike a balance between two extremes: no merging leaves too much duplication in place, while too much merging makes the polygons too big. Our compromise is to impose a restriction on the polygon merging which says that the merged polygon cannot be longer in either direction than the equivalent of 300 pixels at the highest scale for the given LOD. Then the polygon merging criterion becomes that two adjacent polygons can be merged if they have the same feature code and feature name, and if the resulting merged polygon would not violate the 300 pixel rule.

3.4 Database Schema

3.4.1 Goals

The primary concern, as far as the data format, is that runtime performance should be as good as we can make it. By "performance", we mean that the server should be able to pull the requested data out of the database and send it across the network as quickly as possible. Network bandwidth is assumed to be scarce, so we do not want to send too much more information than is needed, although a little more than is specifically requested may improve performance by serving as a de facto prefetch. Since network bandwidth may be low, compressing and uncompressing data is assumed to be faster than sending uncompressed data. Since today's hard drives are large relative to the amount of data that we need to store in practice, disk space is not a major direct factor, but, all else being equal, database performance is better when less disk space is used, so it is an indirect factor.

We are only secondarily concerned with preprocessing speed. We assume that the database is created once and is thereafter read-only, so we do not need to accommodate dynamic updates. In reality, geographic databases do need to be updated as new roads are built, errors are discovered, etc., but these events are infrequent enough that it would suffice to, for example, build a new database once a week, and then simply switch the runtime system over to the new database and remove the old one (or archive it). Preprocessing performance is a concern only in so far as we want the database to be buildable in no more than a few days; improvements beyond that are certainly desirable, but they should not come at the expense of runtime performance.

At runtime, the client will be making requests for all data within a particular geometric range and at a particular scale. Though not required, it is allowable

to partition the infinite number of scales into a finite number of discrete levels of detail (LODs). The amount of duplicate information that needs to be sent to the client should be minimized. The schema must include information that allows the server to track in the cache copy which data elements the client currently has, and execute the pruning algorithm.

3.4.2 The Schema

In our database schema, we use eight predefined LODs. All data for a given LOD is stored in a single table, and each LOD is queried separately from the others. Hence, there is some duplication of data across LODs, but the performance of a query on one LOD is not affected by the data in another. This is important when we consider that in the final database, Level 0, which has the highest level of detail, requires more than 1 GB for the table, and about 91 Mb for the index, while the remaining seven LODs *combined* require just 62 Mb for the tables and 11 Mb for the indexes. Levels 2 through 7 require under 19 Mb for the tables and 3.4 Mb for the indexes. Level 3, for example, requires about 4 Mb for the table and less than 1 Mb for the index. Clearly a range query on the Level 3 table is going to be much faster than one on the Level 0 table. Since much of the navigation in a user session will typically be done at levels 1 through 7, the performance is quite a bit better than could be had with all LODs in a single table.

The data in each LOD is organized around the polygon containment hierarchy. Recall from section 3.3 that after merging the features are partitioned according to which top-level polygon they intersect with, assuming that border lines are duplicated and associated with two top-level polygons. For storage in the database, every feature—line, landmark or polygon—that intersects a given top-level polygon is listed and compressed into a byte array, or blob. It is these blobs that are stored in the database. There are eight tables, one for each LOD: `mapblobs0`, `mapblobs1`, ..., `mapblobs7`. Each record in each table has three fields: the blob; a database-wide unique integer blob id; and the blob's bounding box, which is the MBR of the top level polygon associated with the blob. An R-tree index is created on the bounding box column.

Table 3.1 shows the number of blobs and the average size of each blob, at each S-Level. Table 3.2 lists the number of geographic objects of each type at each level in the intermediate database, which will be described in the next section. This gives a rough idea of how many objects are in each blob, but it isn't exact because most top-level polygon border lines are duplicated in the blobs. We are

	num blobs	avg. size (bytes)
S-Level 0	1,356,216	1,442
S-Level 1	115,596	1,946
S-Level 2	35,592	1,174
S-Level 3	10,987	760
S-Level 4	2,238	482
S-Level 5	385	419
S-Level 6	74	499
S-Level 7	1	8,782

Table 3.1: Blob Database Statistics

also interested in how much overlap there is among the blob bounding boxes, because it gives a good idea of how many blobs will be returned per query. As an estimate of this, we summed the areas of the blob bounding boxes at S-Level 0, and it came to 4,860,538 square miles. The true area for the dataset should be 2,962,081 square miles, which means our computed value is 64% over. If we assign all of this increase to overlap, ignoring the fact that blob bounding boxes on the edge of the geography reach out into the ocean, then it suggests that the average number of blobs returned by a query should be about 1.64 at S-Level 0. At the higher levels such a computation becomes less meaningful because the edge effects become much greater.

Within a blob, each line is represented as line id, feature code, feature name, number of points, coordinates for each point, and a boolean flag indicating whether the line is a state border. Each landmark is represented as landmark id, feature code, feature name, and point coordinates. Each polygon is represented as polygon id, parent polygon id, feature name, number of border lines, integer array with the line id of each border line, boolean array with the orientation (forward/backward) of each border line, boolean flag indicating whether the polygon is water, and boolean flag indicating whether the polygon is a park. The polygons do not store an explicit feature code; instead the class of a polygon is deduced from whether it is water, whether it is a park, and whether it has a name.

3.4.3 The Intermediate Database

With the database schema described in section 3.4.2, the server never needs to concern itself with individual geographic features; the blobs are pre-packaged, compressed, and ready to go. It must store in the cache copy only blob ids and

bounding boxes, and cache pruning is done at blob granularity. In an earlier version of the system, we used a different schema which was not based on blobs. Since that schema is still used by the preprocessor as an intermediate stage in the creation of the final database, it is described in detail here.

This schema has "permanent" tables that are needed by the runtime system, and "temporary" tables that are used only during preprocessing and can then be discarded. Furthermore, the permanent tables contain some temporary fields that are only used during preprocessing. This section only covers the permanent fields of the permanent tables; the temporary tables and temporary fields are discussed in section 3.5.5. There are eight permanent tables each for lines and polygons, associated with the eight S-Levels. The tables are called `lines0`, `lines1`, ..., `lines7`, and `polygons0`, `polygons1`, ..., `polygons7`. In theory we could also have eight tables for landmarks, but in our current system we only use landmarks at S-Level 0, so we only need one table for landmarks, called `landmarks`. Each of these tables has an R-tree index on its range field (see below). We also have one table, called `db_summary`, that has only one record with only one field: the bounding box of the entire dataset. Table 3.2 shows how many lines, polygons, and landmarks are in the database at each level (after merging).

Each record in the `lines`x table corresponds to one line, after merging. It has eight permanent fields: (1) the line id, (2) the feature code, (3) the feature name, (4) a boolean flag for whether the line serves as polygon border, (5) a boolean flag for whether the line is a state border, (6) the number of points (endpoints and shape points are not distinguished in the database), (7) an array of point coordinates, and (8) a range. For a line that is not a polygon border line, the range is the bounding box of the line. For polygon border lines, the range is the minimum bounding box of the union of the polygons that the line borders. The reasoning behind this is that at runtime, when the client sends a range request, it is possible that a polygon will intersect the request range, but a line on its border will not. But if a polygon is sent to the client, it can only be reconstructed by the client if all of its borders are also sent. So the range for each border line is expanded so that it will be sent to the client as part of any response that includes polygon that it borders.

Each record in the `polygons`x table corresponds to one polygon, either top-level or child, after merging. It has eight permanent fields: (1) the polygon id; (2) the id of its parent; (3) the feature name; (4) an array of line ids for the border lines, in order; (5) an array of boolean values, for whether each line is oriented forward or backward; (6) a boolean value for whether the polygon is a water feature; (7) a boolean value for whether the polygon is a park; (8) a range

	num lines	num polygons	num landmarks
S-Level 0	21,531,051	1,834,576	241,192
S-Level 1	3,005,328	582,269	0
S-Level 2	378,792	109,873	0
S-Level 3	81,425	25,946	0
S-Level 4	11,996	4,271	0
S-Level 5	2,614	635	0
S-Level 6	871	117	0
S-Level 7	416	20	0

Table 3.2: Numbers of geographic objects in intermediate database, after merging, at each S-Level

	total	tables	indexes
final database, 300 pixel merge limit	1,243	1,141	102
intermediate database, 300 pixel merge limit	8,492	6,616	1,877
intermediate database, unlimited merge	5,757	4,468	1,289

Table 3.3: Database Sizes, in Mb

representing the polygon's MBR.

Each record in the `landmarks` table corresponds to one landmark. It has four permanent fields: (1) the landmark id, (2) the feature code, (3) the feature name, (4) a range. Of course the range of a landmark is degenerate, covering only a single point. The only reason this field is a range type instead of a point type is that PostgreSQL does not currently support R-tree indexes on point data.

The superiority of the final database can be seen in the size of the two databases. Table 3.3 compares sizes for three databases: the final database, which uses the schema with the blobs, and is based on a polygon merging limit of 300 pixels; an intermediate database, also with a polygon merging limit of 300 pixels; and an intermediate database with no polygon merging limit. All values are in Mb. The total size is also split into size for tables and indexes; values might not add up because of rounding. Comparing the final database to the intermediate one with the same merge limit, the final database is only 14.6% as big, and the indexes are only 5.4% as big. Between these two databases, a given query should return an amount of data that is about the same percentage of the whole dataset, and in that case the number of pages read from the tables for a query should depend linearly on the size of the tables. For searching the index,

the relation between number of pages read and size of database is not so clear—we would normally expect a logarithmic relationship for a tree-based structure, but a search in an R-tree frequently needs to search more than one subtree from any given internal node. In any case, smaller indexes will in general mean fewer pages read. So the final database should give much superior performance to the intermediate one, and some very preliminary and informal testing has borne this out.

3.5 Processing the TIGER Files

The preprocessor has three distinct phases. In the first phase, the raw TIGER data is converted to a more convenient and space efficient internal format and stored in the database in (mostly) full detail, uncompressed and not packaged into blobs. In the second phase, which is executed multiple times, the higher, more generalized and simplified LODs are successively generated, each from the LOD below it. In the third phase, the now-complete database is packaged into pre-compressed blobs and stored in a second database that will be used at runtime. This section describes the protocol that we have developed for the first phase, and the other two phases are described in the next two sections.

3.5.1 Goals and Obstacles

The raw TIGER data that we get is unsuitable for use in the runtime system for several reasons. Most obviously, it is spread out over many thousands of text files, with no indexing. Multiple files have to be read just to get the full description of a single feature. We do not use many of the descriptive data fields. Each county is treated separately, making it difficult to recognize cross-county features, which is especially necessary when we generalize for the higher LODs. There is duplication far in excess of what we need, as discussed in section 3.3.

In this first phase of preprocessing, therefore, we want to convert the data into a format that can more conveniently be used by the runtime system, and that can be used by the preprocessor as a base for building the LOD hierarchy. More specifically, our goals include the following:

1. Eliminate the duplication of county border lines, and unify the multi-county data into a single dataset.

2. Eliminate much of the duplication within a county by merging together adjacent polygons and polylines.

3. Deduce the polygon containment hierarchy, which is necessary for runtime display, and make it consistent across counties.

4. Save various pieces of information that will be needed later when building the LOD hierarchy: for polygons, the county that it came from; for lines, counties for the left and right sides and polygon ids for the left and right sides; for landmarks, the county that it came from.

5. Keep the number of disk accesses low enough so that the preprocessing doesn't take more than a few days.

All of this must take into account that in general no more than one county may fit into main memory at a time. The actual protocol for doing this is given in section 3.5.5, but we look in detail at the two main issues that make the process difficult: deducing the polygon containment hierarchy, and processing the county border lines.

3.5.2 Deducing the Polygon Containment Hierarchy

Much of the complexity of the process is involved in deducing the polygon containment hierarchy, while dealing with the anomalies that were described in section 3.2.3; specifically, in determining the parent for each polygon in the case where the polygon and its parent are in different counties. We cannot know ahead of time which county will be processed first, the one with the parent or the one with the child. And in the case of Prince William County and Manassas, Virginia, one county can have both parent and child polygons with respect to the other county.

To help deal with these anomalies, define a *cluster* as a connected set of polygons having the same parent, and a *family* as a connected set of clusters having the same parent. (The parent of a cluster is the parent of each of its polygons.) The clusters are maximally connected within each county—i.e., within a county, two clusters that have the same parent cannot be adjacent. Therefore, two clusters in the same family will always be in different counties. Two clusters from different counties but with the same parent will only be in the same family if they are adjacent—they need not be. So, a single parent polygon may be associated with multiple families, a single family may have multiple clusters, and a single cluster may have multiple polygons. Consider figure 3.9. Counties, A, B, and C each have all their polygons in a single cluster, numbered 1, 2, and 3, respectively. Clusters 1, 2, and 3 the same parent, polygon P1, and are therefore

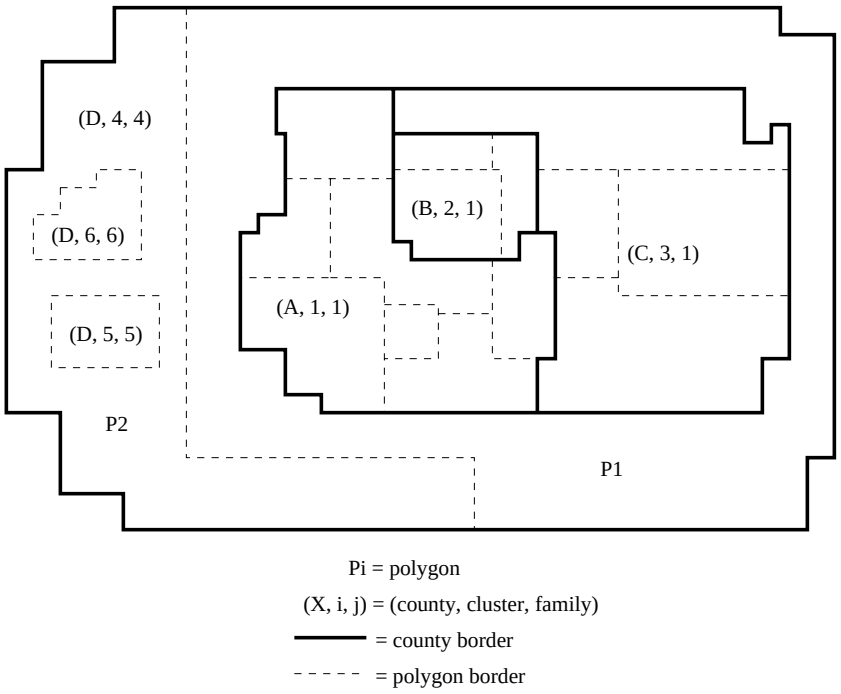

Figure 3.9: Cluster and Family Diagram

in the same family, numbered 1. (The family number is always the same as the number of one of the clusters in the family.) County D has three clusters. Cluster 4 contains polygons P1 and P2; this cluster is in family 4. (The parent of the polygons in cluster 4 is not indicated, and is not important for this discussion.) Each of clusters 5 and 6 has one polygon, is in its own family, and has parent P2. Now we need a way to compute the clusters and families, without ever having more than one county in memory at a time.

Below we'll see that we can partition the polygons in the current county into three classes: (1) those whose parent is known to be a polygon in another county, (2) those whose parent is known to be a polygon in this county, and (3) those that are *county level*, which means that they are assumed to be top-level, until proven otherwise. For example, if Manassas is processed before Prince William, then while processing Manassas its county level polygons are assumed to be top-level. But when Prince William is processed, we'll find out that Manassas is actually inside a large merged polygon in Prince William. Another example is Manhattan (see figure 3.2). If Manhattan is processed before Hudson County, New Jersey, then it will have three county-level clusters, but if Hudson County is processed

first, then Manhattan will have one county-level cluster, and two clusters that are known to have parents in a different county. (In either case it may also have clusters that are inside a polygon within Manhattan.)

For now suppose that this partitioning has been made for the current county. Within each class, polygon adjacencies are traced to compute the clusters, and each cluster falls into one of the three classes. Each cluster is initially assigned to a family with the same id number as the cluster; later the clusters will be grouped together into families using standard union-find techniques (see [20], or many other books on algorithms). At this point we know the parents for polygons in the first two classes. We don't know the parents for polygons in the county-level clusters, but we do know that all polygons in a given county-level cluster must have the same parent. What we do is temporarily set their parent fields to null, meaning that they are top-level, but also note in the database what cluster they belong to, and we also keep a mapping of clusters to families. That way if we later discover that this family has a parent, then we'll know which polygons to update.

Computing the three classes, as well as recognizing clusters in class (1), those that have a parent in another county, all comes down to being able to recognize polygon holes. A polygon that is inside the hole of a polygon in another county is in class (1); a polygon that is inside the hole of another polygon in the current county is in class (2); all other polygons are in class (3). Furthermore, if a polygon in this county has a hole, and the lines on the border of the hole have a null field for the polygon id on the other side, meaning that the polygon on the other side is not in this county, then that information must be saved properly in the database so that when the counties that are inside the hole are later processed, their polygons will be properly recognized as being in class (1) at that time.

Recognizing the holes is part of the process of constructing the polygons from information in the lines. Each line stores the polygon id on its right and left sides. A null polygon id on one side indicates that the line is on the county border. To construct a polygon, we first gather together all lines that have that polygon on one or the other side, and then group the lines into loops by matching endpoints. A polygon without holes will have only one loop. The loop whose MBR contains all the other loop MBRs must be the outer boundary of the polygon; any other loop is a hole. Lines in an inner loop are called *hole borders*. Polygons from the current county that are on inside of a hole border can be put into class (2). If a polygon on the inside of a hole border is from a county that was previously processed, then it would have previously been assumed to be in a top-level cluster, but now we know that it is not top-level, so we must note in the database that

the parent for its cluster is the current polygon (the one with the hole). If a polygon on the inside of a hole border is in a county that has not been processed yet, then we must store this hole border in the database in such a way that when that other county is processed in the future, this hole border will be found and polygons in that other county can be properly placed into class (1).

Consider again figure 3.9. Suppose the counties are processed in the order A, B, C, D, and suppose that each county has only one cluster. After A is processed, cluster 1 is county level. After B is processed, cluster 2 is also county level, and it will be noted that it borders cluster 1, and it will be put into the same family with cluster 1. Cluster 3 in county C will similarly be put into the same family with clusters 1 and 2. When D is processed, its raw TIGER polygons are merged into a single polygon P, and within P there are hole borders that border polygons in clusters 1 and 3, so it will be known at that time that the family that those polygons belong to has parent P. The necessity of the clusters can be seen from the fact that not all polygons in counties A and C (clusters 1 and 3) share a border with P, which means that it is not sufficient to merely update the parent of *polygons* that share a border with the hole. The necessity of families can be seen from the fact that no polygon in cluster 2 shares a border with P, so that it is insufficient to merely update the parent of *clusters* that share a border with the hole. By using clusters and families, all polygons can be properly associated with their parents. The cluster and family information is stored in the database at the end of processing each county, so all this can be done with only having one county in main memory at a time. (Or, more accurately, one county, plus the cluster and family data, plus a small amount of information describing what we have previously discovered about polygons and clusters that border polygons in this county.)

3.5.3 Processing the County Borders

The second major difficulty in preprocessing is in dealing with the county border lines. We want to keep only one copy of each line, but a line on the border of two counties has a copy in each of them. We also want to merge the lines, but recall from section 3.3 that the decision on whether two lines can be merged depends on the polygons on either side of them. For county border lines, we can only know the polygons on both sides when processing the second county. This also indirectly affects the processing of border polygons, since these polygons don't know how many lines make up their borders until the border lines have been merged.

Our solution uses some temporary tables used only during the preprocessing phase. When a county border line is first encountered, it is stored in a temporary file in its raw, unmerged state. When the bordering county is subsequently processed, the border lines that were previously stored are matched with ones from the current border, and then the merging can be performed, since the polygons on both sides of the lines are now known. Finally the merged lines are stored in the permanent lines table. For border polygons, we go ahead and merge them, but store them only in the temporary table. Once all of a border polygon's lines have been merged, by virtue of all the bordering counties being processed, the polygon can be moved into the permanent polygons table, with the new set of merged lines as its borders. Other temporary tables that aid with this process are described in the next section.

The full protocol for processing the TIGER data and producing the S-Level 0 database is in section 3.5.5, but first the temporary tables are described in the next section.

3.5.4 Temporary Data

It was already mentioned in section 3.5.1 that some extra fields are temporarily needed during the preprocessing phase for building the LOD hierarchy. These are the county for polygons, the left and right counties and left and right side polygons for lines, and the county for landmarks. Here we list the temporary tables that are needed during the processing of a particular LOD, to unify the data across counties.

blines Temporary storage for the unmerged lines that lie on a county border. Lines go into here when the first county that they border is processed, and go out when the second county they border is processed. At that point, they can be merged and put into `lines0`. In `blines`, each line has a boolean flag indicating whether the line is a hole border, and the cluster number for the polygon that it borders in that first county.

bpolygons Any polygon that currently has one of its border lines in `blines` instead of `lines0` is in here. Once all of its border lines have been merged and put into `lines0`, the polygon can be moved into `polygons0`. This table contains all the normal polygon fields except for the list of border line ids; that information is kept in `sides`.

sides A record for each line on the boundary of any polygon that is in `bpolygons` will be in here. Contains fields for the polygon id and line id, the index

number of this line in a traversal of the lines around the polygon's boundary (arbitrary starting point), and a boolean flag for whether the line is forward oriented. When it comes time to move a polygon from `bpolygons` to `polygons0`, the polygon's border lines can then be recovered from `sides`. The purpose of this is to make it easy to modify the border line lists of the polygons in `bpolygons`. When two lines from `blines` are merged, the merged line will have the line id of one of the constituent lines; the record for the other line need only be deleted from this table, and no changes directly made to the polygon in `bpolygons`.

cluster_family A mapping of clusters to families, with a unique index on the cluster field.

family_parent Mapping of families to the polygon id of the parent, with a unique index on the family field.

county_line_count Mapping of county numbers to the number of lines from that county currently in `blines`. When the line count goes to zero, all polygons from that county in `bpolygons` can be moved to `polygons0`.

3.5.5 Protocol for Producing S-Level 0

The process of producing S-Level 0 starts by executing the following operations on each county:

1. Read the raw TIGER data from the zip file.

2. Match the border lines in this county with border lines from previous counties in `blines`, removing the matched lines from `blines`, and noting which lines are hole borders and the adjacent cluster numbers.

3. Retrieve relevant information from `cluster_family`, `family_parent` and `county_line_count`.

4. Compute the polygon and line merge sets, generate the merged line objects, and update `sides` to reflect the merged border lines.

5. Generate the polygon objects, finding holes and computing new cluster and family information, as described in section 3.5.2.

6. Store lines in `lines0` or `blines`, polygons in `polygons0` or `bpolygons`, landmarks in `landmarks`, and add records for the new border polygon sides in `sides`. Compute the area for each polygon using a triangulation/line sweep algorithm [57].

7. Update `cluster_family`, `family_parent` and `county_line_count` tables. If any county's line count has dropped to zero, then all polygons from that county are moved from `bpolygons` to `polygons0`, and its tuples in `sides` and `county_line_count` are deleted.

After all counties have been processed as above, we are left with the outer border lines in `blines`, and their adjacent polygons in `bpolygons`. Because these will not be bordering any other counties, they must be processed a separate way, the major difference being that adjoining lines can be merged if they agree in the polygon on only one side (the other side has no polygon). Here is the method:

1. Select *all* lines from `blines` and merge them, using just the polygon id on one side to determine which can be merged, and update `sides` to reflect the merged lines.

2. Store the merged lines in `lines0`.

3. Move all polygons from `bpolygons` to `polygons0`.

4. Store the bounding box of all the lines into `db_summary`.

Finally, we are ready to finalize the parent fields in `polygons0`, using information from `cluster_family` and `family_parent`. Recall that polygons in county level clusters have their parent fields set to null, indicating that they are top-level, but some of them are later discovered not to be top-level after processing more counties. The correct parents for these polygons is embedded in the `cluster_family` and `family_parent` tables. Each polygon record contains the polygon's cluster number, the family is then found from the `cluster_family`, and the parent is found from the `family_parent` table.

3.6 Generalization and Building the LOD Hierarchy

Generalization of geographic data is, as Weibel says, "... a complex process with ill-defined objectives, involving a good deal of subjective decisions" [104].

Nonetheless, we have a mostly automatic approach that gives adequate results for our purposes. This process involves three parts: choosing the range of scales to associate with each LOD, choosing the set of features to include in each level, and extending the line and polygon combining across county borders, to find more opportunities for duplication elimination and simplification.

For choosing the scales to associate with each LOD, we use the pixel size as the reference unit of length, rather than the inch or centimeter, since the applet may display on screens with different resolution. Let *pixel level of detail*, pLOD, be the number of meters per pixel. We currently use eight hand-chosen pLOD's, $0, 14, 75, 200, 700, 2500, 7500$, and 22000 as the breaking points for levels of detail. This defines eight *simplification levels* or S-Level for short: S-Level 0 corresponds to pLOD's $[0, 14)$, S-Level 1 corresponds to pLOD's $[14, 75)$, ... S-Level 8 corresponds to pLOD's $[22000, \infty)$.

3.6.1 Manual Generalization

The TIGER system gives some information about feature type that can be used for automatic feature selection when generalizing the data. For example, distinctions are made between major highways, secondary roads, local roads, etc. However, in some cases the distinctions are rather crude. For example, in New York, 42nd Street and Thompson Street are given the code A41, meaning local road, but anyone who has even visited New York (or who has some familiarity with Broadway musicals) knows that 42nd Street is a much more important street than Thompson Street. In order to incorporate this kind of knowledge into our system, we have developed a tool, pictured in figures 3.2 through 3.7, for manually choosing features to include in a level of detail. The checkboxes at the right are for selecting/deselecting classes of features. Individual features can be selected with mouse clicks. The user can also simplify the lines, using the Douglas-Peucker algorithm [22], and filter polygons by area.

Although we were able to produce good generalizations with this tool, it turned out to be too labor intensive. It requires individually generalizing more than 3000 counties at seven LODs; plus, it requires knowledge about which streets are important in each of the counties. Therefore, we decided that for this research project we would sacrifice some map quality and use a mostly automatic approach. The only part done by hand is to specify, for each level, the classes of features that will be included, the area of the minimum size of a polygon to keep, and the error bound for line simplification. Table 3.4 shows the choices for each level. The minimum polygon size for each S-Level is equivalent to 20 square

S-Level	line features	min polygon	error bound
0	state borders and all visible lines	0 m^2	0 m
1	state borders, major highways, primary and secondary roads, railroads, perennial water	3,920 m^2	28 m
2	state borders and major highways	112,500 m^2	75 m
3	state borders and major highways	800,000 m^2	300 m
4	state borders	9.8 km^2	1.4 km
5	state borders	125 km^2	5 km
6	state borders	1,125 km^2	15 km
7	state borders	9,680 km^2	44 km

Table 3.4: Generalization Characteristics of Each S-Level

pixels at the lowest pLOD for that S-Level. The line simplification error bound is between one and two times the minimum pLOD for the S-Level. Once these decisions have been made the preprocessor can run without human intervention. 42nd Street, alas, is not included in our S-Level 1 map for New York.

3.6.2 Automatic Generalization

Given the S-Level generalization specifications from the previous section, the preprocessor produces the LOD automatically, assuming that the next lowest LOD has already been produced. Polylines are chosen by their feature code, polygons are chosen by size, and polylines are simplified using the Douglas-Puecker algorithm [22].

When a polygon is dropped because of being too small for a given S-Level, its area is acquired by an adjacent polygon, and it is not clear how to choose which adjacent polygon. Our initial naive algorithm was to give top priority to adjacent polygons with the same name and code, second priority to polygons with just the same code, and all others third priority. Ties were broken by choosing the polygon with the largest area. This method led to some strange effects at the higher LODs; for example, water that was clearly the Missouri River was labeled Mississippi River.

This situation was discussed in [97], where it was proposed that a "collapse

function" be based on each polygon's "weight-factor" or importance (e.g., area), its type compatibility with the polygon being dropped, and the length of the border between them. We have experimented with several functions of the form $f = (f_1, f_2, f_3)$, where $f = f_1$ if the polygons do not share a name or code, $f = f_2$ if they share name or code but not both, and $f = f_3$ if they share both. Let A be the area of the candidate acquiring polygon, and b be the length of the common border between it and the polygon being dropped. Then the functions are as follows:

1. $f = (b, 1.5b, 2b)$
2. $f = (b, 2b, 3b)$
3. $f = (b, 3b, 5b)$
4. $f = (b, 2b, 4b)$
5. $f = (b^2/A, 2b^2/A, 3b^2/A)$
6. $f = (b^2/A, 2b^2/A, 4b^2/A)$

The idea of using b^2/A was originally suggested, in a different context, by Alan Siegel.

Figures 3.10 through 3.12 show the results of each function as applied to the state of Maryland, at S-Level 4. The last two functions, which take into account area as well as border length, are much better at preserving the complicated shoreline of Chesapeake Bay, which is the dark area in the middle. We ended up using the last one, and this avoided the problem with the Missouri and Mississippi rivers that we had with our naive solution.

3.6.3 Grouping the Counties

At S-Levels higher than 0, we are not dealing with the full detail, so more than one county can fit into memory. We also would like to be able to merge lines and polygons across county borders—the more counties we can process at once, the more opportunity there will be for duplication elimination and simplification. But we also want the group of counties that are processed together to be "nicely shaped"; since we limit the size of the MBR of a polygon, a snakelike shape of counties will not give as efficient a polygon merging as a square or circle, for example. So we want to partition the counties into groups that are connected, nicely shaped, and have no more than some given amount of data, so that they

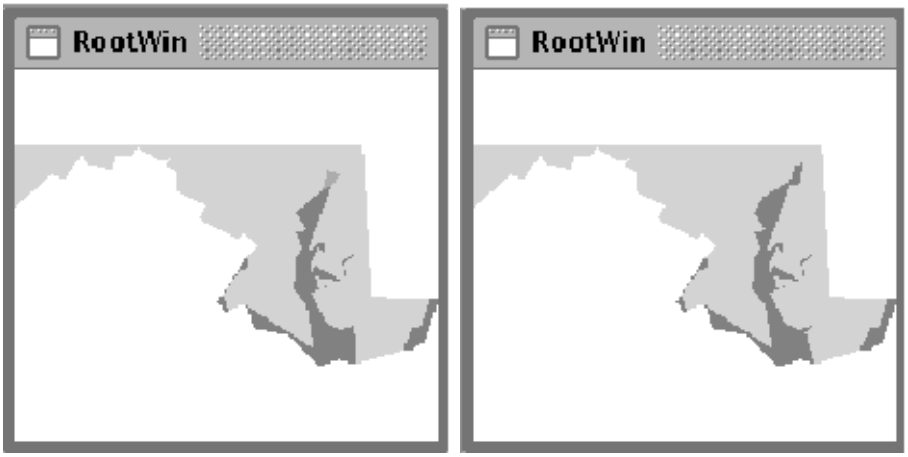

Figure 3.10: Maryland, Area Acquiring Algorithms 1 (left) and 2

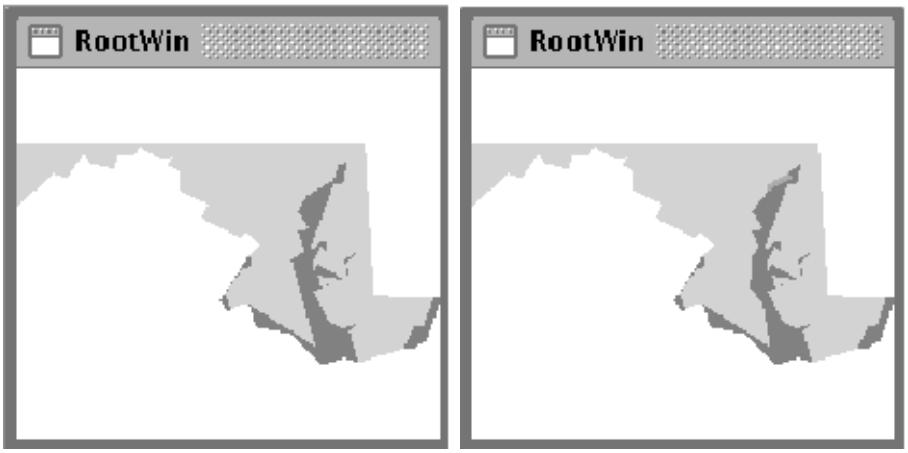

Figure 3.11: Maryland, Area Acquiring Algorithms 3 (left) and 4

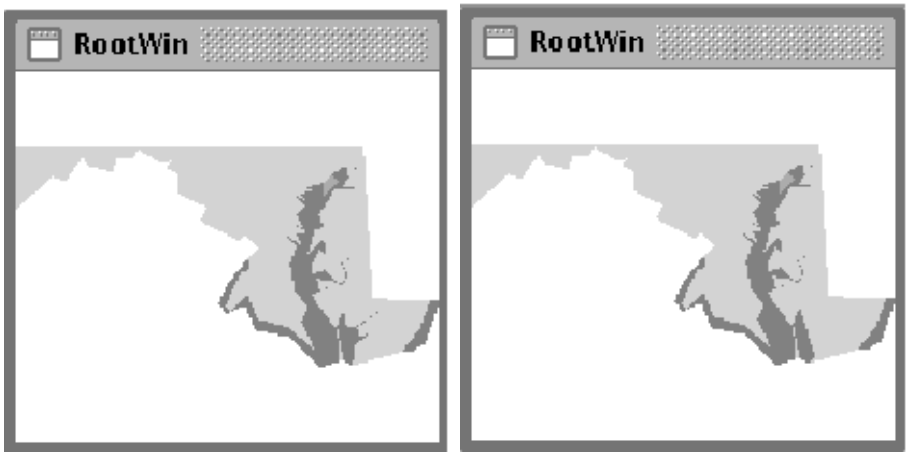

Figure 3.12: Maryland, Area Acquiring Algorithms 5 (left) and 6

will fit into memory. We estimate by the "amount of data" by the total number of endpoints and shape points in all the lines at full detail.

Our solution is to use a simple heuristic that produces nicely shaped groups that obey the size limit, but that might not be connected, and then to modify the grouping by hand to get connectivity. The heuristic is this: pick the axis along which the dataset is "longer", sort the counties along that axis by their range centers, and split the set into two roughly equal sized pieces, where size is number of points at level 0. Recurse until the group contains less than the maximum number of points. When computing the higher levels of detail, use the groups at the next lower level as the "counties".

This heuristic produced groups that were nicely shaped and fairly evenly sized (according to number of points). Unfortunately, the groups chosen were not necessarily connected. For example, two counties in Virginia that are completely contained inside other counties were not grouped with their surrounding counties; Norfolk county in Massachusetts is in three pieces, and the surrounding counties of one of the pieces were put in a different group from the other Norfolk pieces and its other adjacent counties. These problems, and others, were solved by manually regrouping where necessary. In order to speed up this manual regrouping process, we have listed in the appendix a set of constraints mandated by these two types of anomalies. "Singleton constraints" deal with the situation where a county is adjacent to only one other, so these two counties must be grouped together. "Multiple piece constraints" deal with the situation where a county has more than one piece, in which case each piece should border another county in the

group.

3.6.4 Protocol for Producing S-Level i, i > 0

Producing the higher S-Levels is in many ways similar to producing S-Level 0: the data is processed in sections rather than all at once, the border lines and polygon containment hierarchy must be made consistent across sections, and the "raw" input lines and polygons must be merged. Here are the major differences:

- Instead of working with one county at a time, we work with one "group", where a group is a set of counties or groups at the next lower level, as determined by the grouping process described in section 3.6.3. Instead of getting the input data from the TIGER files, we get it from the next lower level in the database, by selecting all objects from that level that come from any of the constituents (counties or lower level groups) of the current group.

- If any polygon is smaller than the minimum area, it is dropped, and one of its neighboring polygons "acquires" the dropped polygon's area, according to the algorithm in section 3.6.2.

- Lines are simplified with the Douglas-Peucker algorithm [22] before being stored in the database. The error bound to use in the algorithm is given in table 3.4.

- The area of each raw, unmerged polygon is taken from its stored value at the lower level, and after polygons are merged the area of the merged polygon is the sum of the areas of the polygons that were merged into it. In other words, the triangulation algorithm for computing area is not used. This is because the line simplification may introduce anomalies into the topology, for example by making a polygon border cross itself, so that computing the area directly is difficult.

- Another topological anomaly that the simplification can introduce is that the range for a child polygon might extend outside that for its parent. For this reason, the value that we store as the range of a border line is not the MBR of all its points. Instead, the range of each raw, unmerged line is taken from its stored value at the lower level, and after merging the range of a merged line is the MBR of the range of each of its constituent lines.

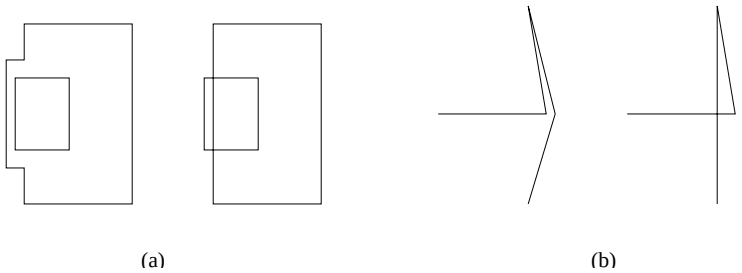

Figure 3.13: Line Simplification Anomalies: (a) child polygon outside its parent, (b) line crossing itself.

Figure 3.13 shows two examples of the topological anomalies that can be introduced by line simplification. While these cause difficulties in the preprocessing, it is acceptable to leave them in the generalized data, since at runtime the goal is visualization, and the error bounds are designed so that the anomalies are off by no more than a pixel.

3.7 Producing the Blobs

The last preprocessing step is to take the data from the temporary database, package it into blobs, compress the blobs and store them in the permanent database. We use the grouping computed for building the LOD hierarchy, and additionally take a default grouping for S-Level 0 in which each county is a group. The process is the same for every S-Level; consider S-Level i. For each group, select from the S-Level i tables in the temporary database all the landmarks, lines and polygons in the group (group or county is a field in each tuple), using the R-tree index on the range to speed the query. Then use the top-level polygons—those that have no parent in the polygon containment hierarchy—as the basis for forming the blobs. A top-level polygon, all its child polygons, all lines with any of those polygons on at least one of its sides, and (almost) all landmarks that intersect the polygon's range are all included in the blob, and the blob is compressed and stored in the new database. Since a landmark can intersect more than one polygon range, the landmark ids are stored in a hash table, and removed from the hash table when they are included in a blob, so each landmark is only in one blob. Blob border lines are generally duplicated, unless they border only one top-level polygon (i.e., the lines on the outer border of the entire data set).

Chapter 4

Visualization Interface

4.1 Related Work

4.1.1 Multi-Scale Navigation

A survey and taxonomy of zooming/panning interfaces can be found in [70]. There, seven types of "image browsers" are delineated:

Detail only browser. This is a single window with panning, usually by scrollbars, and no zooming. This is easy to implement and common in many programs, but it is only useful when the zoom factor is quite small.

One window with zoom and replace There is one window and upon any zoom or pan request a new image replaces the old one. This is the method used by MapQuest [21] and other web map servers. It has the major problem that the user is not able to see context and detail at the same time.

Single coordinated pair (detail and overview) In a typical setup a small window has the overview and a large one the detailed view. The overview often shows a "field-of-view" rectangle for the current location of the detail view. The WebTOC (web table of contents) [58] is a version of detail and overview where the overview is a hierarchical table of contents for a web site, and the detail is an individual web page. Shneiderman [80] has noted that for zoom factors greater than 30 simple overview and detail becomes inadequate, and intermediate views are needed.

Tiled multilevel browser This extends the detail and overview to allow one or more intermediate views between the global and detail views, in sequence.

Each view is linked to the next in the sequence with a field-of-view rectangle.

Free zoom and multiple overlap The user may "specify, move, reshape and delete every window as they want. Any side by side comparison is possible." There is no linking or coordination among the windows; each is independent. This gives the user great freedom, but also requires window management, which can be time consuming.

Bifocal view browser This is a variant of detail and overview in which the detail is like a small magnifying glass that can be moved over the overview. See [82].

Fisheye view The fisheye view, first introduced by Furnas [34], shows only one view, but it is distorted so that the area of interest is shown at higher magnification than other areas. A survey of distortion-based multi-scale views, mostly fisheye variants, can be found in [51]. Schaffer, et al. [75], also give a survey, and introduce a fisheye variant in which the view may have multiple focal points.

The fisheye view is actually an example of a broader class of visualization techniques called focus+context. Another approach that used distortion to mix multiple scales in a single view is the hyperbolic browser [50, 49]. In focus+context screens [2, 1] an LCD monitor showing a high resolution focal image is embedded in a wall-sized display upon which the low resolution context image is projected—no distortion, but obviously special hardware is required. None of these techniques is adequate for the large zoom factors (around 10,000) that we have to deal with.

An interesting related visualization interface is used in Pad and its successor Pad++ [69, 5, 3, 4]. This is single view, but infinitely big with infinite zooming. "Portals" provide links from one part of the pad to another, which helps with giving context. But these systems are more appropriate for visualizing collections of disparate objects, rather than a unified map.

Some of the papers referenced above have been collected in [9], with commentary.

4.1.2 Labeling

Another issue that must be dealt with in visualization of geographic information is feature labels. A brief but good survey and taxonomy of labeling techniques can be found in [23]. Traditionally the problem in cartography has been *static*

labeling, in which label positions are pre-computed once and for all, and as many labels as possible are shown without too much overlap. (Finding an optimal solution is NP-hard.) Christensen, et al., [16] ran some experiments with different techniques, and found that a simulated annealing solution that they developed gave the best results, but other solutions ran faster. Static techniques, however, are not a good fit for interactive exploration, partly because of the computation time, and partly because a static solution cannot possibly anticipate all of the viewpoints that an interactive user will traverse.

The dynamic labeling techniques listed in [23] include the following:

Infotip Also known as "cursor sensitive balloon label". A label for a feature appears when the mouse passes over the feature, and disappears when the mouse moves away. Normally the label appears near the feature, but it might also appear in a side window.

Temporal brushing As with infotip, a label appears when the mouse passes over a feature, but the label remains on the screen as new labels come up. Overlap is allowed. See [17].

All or Nothing All labels appear when the number of objects on screen is below a given threshold. An example is the starfield display of Ahlberg and Shneiderman [8].

Focus+context The bifocal and fisheye browsing techniques mentioned above can serve as the basis for labeling, where the magnifying glass (magic lens), or magnified focal area, is expanded enough to make room for labels (or might contain only labels).

Sampling In [10], the labels are spread out over time. Three labels are shown when the display is at rest, and one when it is in motion. Every second, the oldest label is removed, and a new one, chosen randomly, is displayed. Objects that are "close" are sampled more often than those that are "distant", and the sampling rates are affected by the history of user sessions.

Excentric labels These are introduced in [23]. A circle is drawn around the focus region, and all objects in the focus region are labeled. However, the labels appear outside the circle, on either side, and in list form so there is no overlap. A line or pointer connects each label to its object within the circle.

4.2 Our Windowing User Interface

For multi-scale navigation, we have chosen a restricted form of the "free zoom and multiple overlap" approach defined in [70]. We give the user most of the freedoms of flexible window management associated with that approach, but combine it with several linking relationships among the windows. These links are described in detail in section 4.2.2. The multiple views are logically organized into a tree by the overview-detail relationship. The client starts by showing only the root window, with the highest overview available, and the user can then open intermediate views and detail views in any tree structure. Figures 4.1, 4.2, and 4.3 show two sample sessions. In session 1 (figure 4.1), the user has opened three detailed views into the root window overview, showing the New York, Los Angeles, and Chicago areas (windows 1, 2, and 3), at scales that are still too zoomed out to see any roads, for example. Then the user has opened a detail window on the New York view (window 1.1), turning the previous New York detail window into an intermediate view. In session 2 (figure 4.2), the user has opened a detail view of the New York area (window 1), then a more detailed centered on Manhattan and Queens (window 1.1), and then two detailed views showing parts of Manhattan and Queens (windows 1.1.1 and 1.1.2). Figure 4.3 shows a session with overlapping windows.

The current dataset, in which the scale changes by a factor of almost 10,000 from overview to highest detail, can comfortably be explored in three overview-detail steps, in addition to the root window overview.

4.2.1 Anatomy of a Window

Each window can be independently zoomed, panned, moved, resized, iconified, and maximized, with the exception of the root window, which can only be moved. Detail windows can be closed, but intermediate view windows and the overview root window cannot. A detail view window can be opened for any existing view window. Here is a list of the graphical objects on each window, and what purposes they serve:

1. *The map.* The largest portion of each window, this is the place for panning, opening new detail windows, re-centering detail windows, and viewing feature labels, and it also indicates the location of each of its detail windows.

    - Drag with the left mouse button to pan in the direction of the drag.

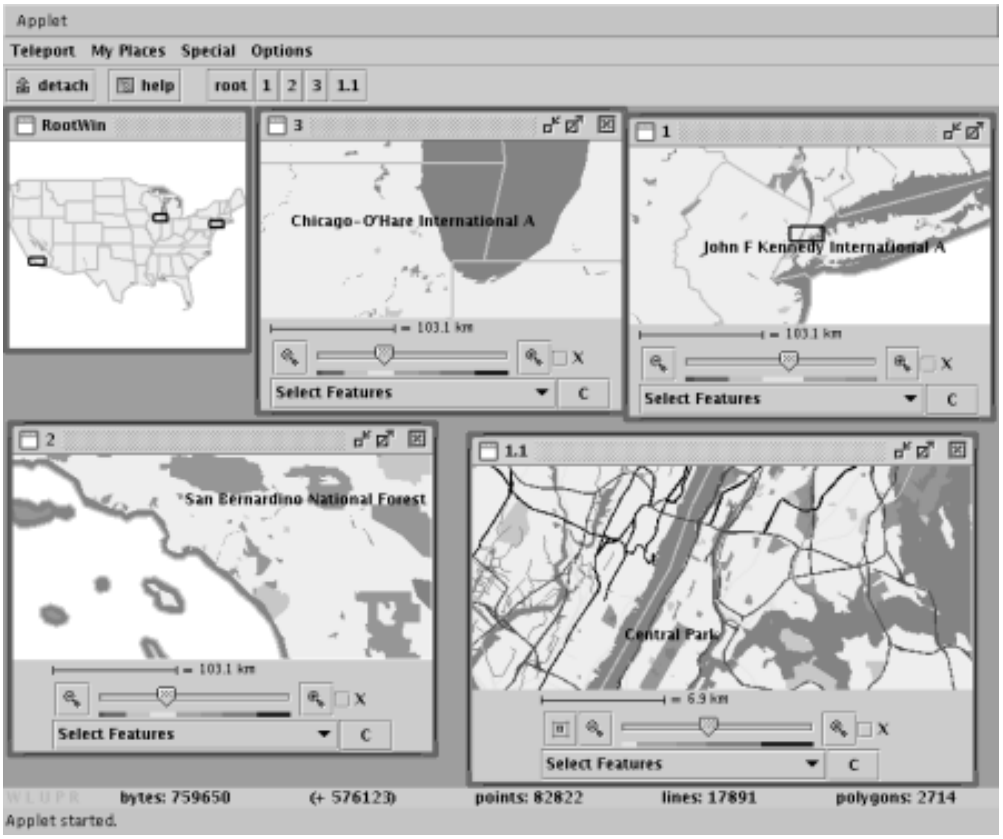

Figure 4.1: User Session 1

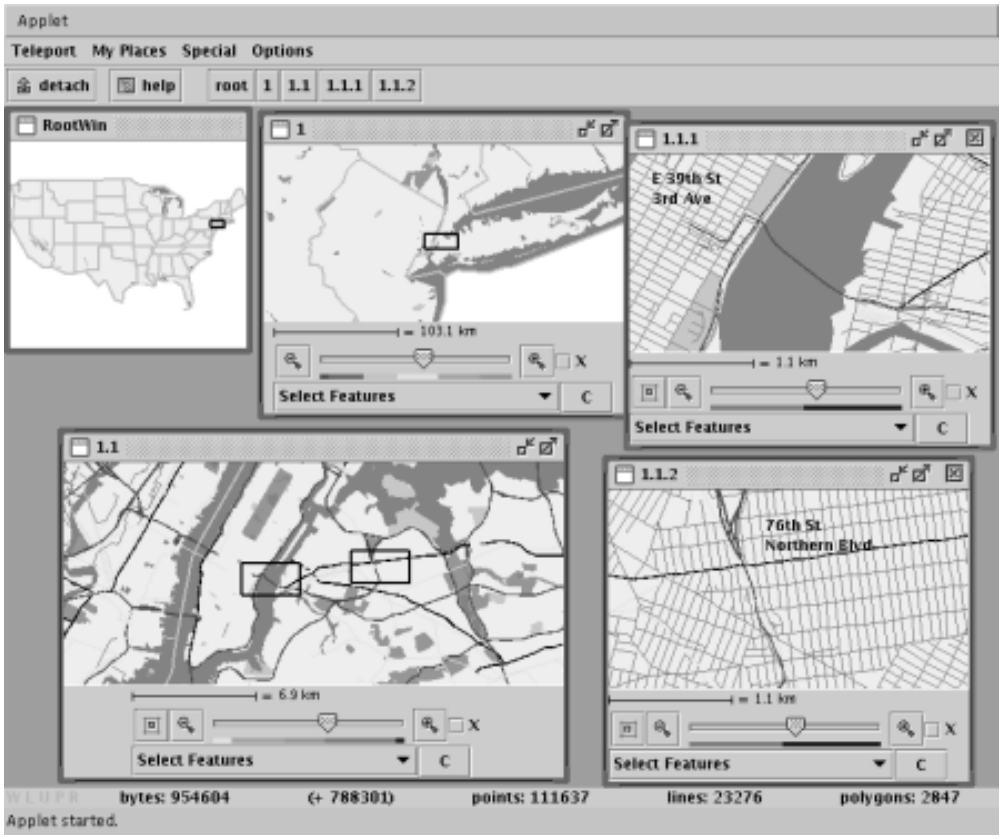

Figure 4.2: User Session 2

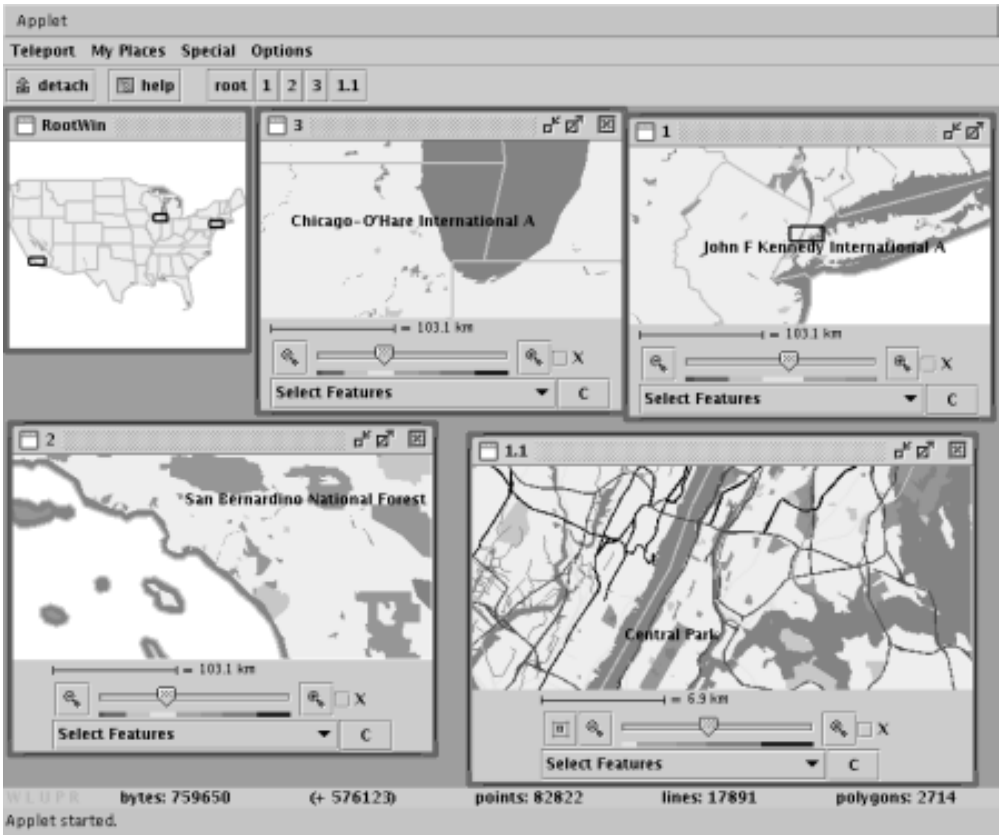

Figure 4.3: User Session 3

- Click with the right mouse button on any point of the map to open a new detail window centered at that point.
- Click with the left mouse button on any point of the map to recenter a detail window at that point. If this map is overview for more than one detail window, then a popup menu will appear from which one of the detail windows can be chosen for re-centering.
- Whenever the mouse pauses over a feature, a label with the name for that feature will appear (assuming the feature has a name). If the mouse is near more than one feature, then multiple labels will appear, one per line. See, for example, window 1.1.2 in figure 4.2; the mouse pointer, though not visible in the picture, had paused near the corner of 76th Street and Northern Boulevard. This labeling technique is called "infotip", or "cursor sensitive balloon label", in the terminology of Fekete and Plaisant [23]. In the current version only one label is shown at a time in a given window, so any previous label disappears when the new one pops up.
- If a map has detail views opened, then black rectangles appear that indicate the location of each detail view. See, for example, the root window and window 1 in figure 4.1.

2. *Iconify and Maximize icons.* In the upper right corner of every window except the root window appear icons for iconifying and maximizing the window.

3. *Close icon.* In the upper right corner of any window that does not have a detail view opened (i.e., is currently a leaf in the tree), appears an icon for closing the window.

4. *Scale display.* Directly below the map is a line 100 pixels long and a distance in meters or km, indicating the current scale of the map.

5. *Zoom slider.* The slider underneath the scale display is used for zooming. It is divided into 100 ticks. The maximum scale, associated with the leftmost tick, is the scale of the parent/overview map; the minimum scale is the maximum of the scales of any children/details, or 250 m per 100 pixels if there are no children.

6. *Zoom buttons.* On either side of the zoom slider is a zoom button—zoom out on the left and zoom in on the right. Pressing a zoom button is equivalent to

jumping ten ticks on the zoom slider, or going to the minimum or maximum scale, if the current scale is less than ten ticks from the edge.

7. *Color bar.* Below the zoom slider, and pointed to by it, is a color bar. Each color represents a level-of-detail. Moving the slider within a color field merely changes the size of the features displayed; moving between colors changes the level of generalization.

8. *Center parent button.* On windows that are not children of the root window, the center parent button lies to the left of the zoom out button. For example, in figure 4.1 only window 1.1 has a center parent button; in figure 4.2 windows 1.1, 1.1.1, and 1.1.2 all have center parent buttons. All other windows in the two figures are either root windows or children of a root window. The purpose of the center parent button is to recenter the parent window on the center point of the child window. This is helpful sometimes when after panning the child away from the parent's previous location, we want to move the parent to the child's current location.

9. *Cross hair checkbox.* To the right of the zoom in button is a checkbox. When this is selected, horizontal and vertical lines appear across the center of the window, which helps in pinpointing locations. In figure 4.4, window 1 has the cross hair selected.

10. *Select features drop-down menu.* Below the color bar is a drop-down menu that is initially labeled "Select Features". This menu contains a list of all of the named features that are in or near the current window location. (Specifically, all named features whose minimum bounding box intersects the current window range.) When a feature is selected, it is highlighted on the map by painting it a special color, and thickening it if it is a line. Features remain highlighted until they are explicitly cleared.

11. *Highlight clear button.* Next to the select features drop-down menu is a button labeled "C". Pressing this button clears all highlights that were selected, returning the features to their normal color and lines to their normal thickness.

4.2.2 Relationships Between Windows

Each overview is linked to its details in several ways:

1. Each overview shows a black rectangle around the area covered by each of its detail views. The appropriate rectangle is automatically updated when the detail is zoomed or panned.

2. If the overview has one detail window opened on it, then clicking the left mouse button on a point on the overview will cause the detail window to be re-centered on that point. If the overview has more than one detail window, then clicking the left mouse button on a point on the overview will cause a popup menu to appear, through which the user can choose which of the details will become re-centered on that point.

3. The *center button* allows a window to recenter its overview (unless the overview is the root window, which never changes).

4. If an overview has one detail window, then the minimum scale allowed for the overview is equal to the current scale of the detail. If an overview has multiple detail windows, then its minimum allowed scale is equal to the maximum of the current scales of its detail windows. (The minimum scale of a detail window with no detail children is 2.5 meters per pixel.)

5. The maximum scale allowed for a detail is the current scale of its overview. (The maximum scale allowed for a window that has the root as its overview is the scale of the root, which is 26.4 km per pixel.)

This linking of allowable scales for parent and child was originally suggested by Zilin Du.

4.2.3 Global Options

The main applet window offers several features worth noting.

1. In the main menu bar, the *Teleport* and *My Places* menus both offer a list of places that, when selected, cause the window that currently has input focus to jump to the given place. The Teleport menu gets its list of places from the map server. The individual user cannot modify it, but the list is in a plain text file, so that we, the maintainers of the server, can easily modify it. We plan to eventually list thousands of cities there. The My Places menu contains places that each individual user can define; these places are stored in the user's browser cookies. Figure 4.4 shows the "Place Editor" window, which is opened by the first option in the My Places menu.

2. The *Special* menu contains options for debugging that are not really meant for the end user.

3. The *Options* menu contains an option for turning labels on and off. Labels are on by default; if they are turned off, they will not pop up when the mouse pauses over a feature. This menu can eventually be extended to more user options, such as selecting/deselecting feature layers—e.g., roads, railroads, etc.

4. The *detach* button causes the entire applet to be transferred out of the web browser window and into its own applet window. That window can then be resized as the user pleases.

5. The *help* button opens a window with brief instructions for using the applet.

6. To the right of the help button are several buttons, one for each window that is currently open. Clicking a window's button causes the window to get the input focus; become deiconified, if necessary; and move to the front, if it is currently covered by other windows.

7. At the bottom of the applet window are several counters, showing the number of bytes, points, lines, and polygons that have been downloaded so far. The number between bytes and points represents the number of additional bytes that would have been downloaded if the data had not been compressed. This information is used largely for debugging.

8. Though it is difficult to see in the figures, there are five letters to the left of the bytes count: "W L U P R". These letters give the user information about what the program is currently doing. When the program is at rest, the letters are a shade of gray that is similar to the background, which is why they are difficult to see in the figures. The letters light up in bright red when the program is active in various activities. Each letter is associated with one activity:

    - **W** indicates that the client is waiting for the server. Specifically, it means that at least one request has been sent to the server that has not been responded to yet.
    - **L** indicates that the client is currently loading a blob from the network.
    - **U** indicates that the client is currently unpacking a blob (i.e., uncompressing it and storing its contents in the cache).

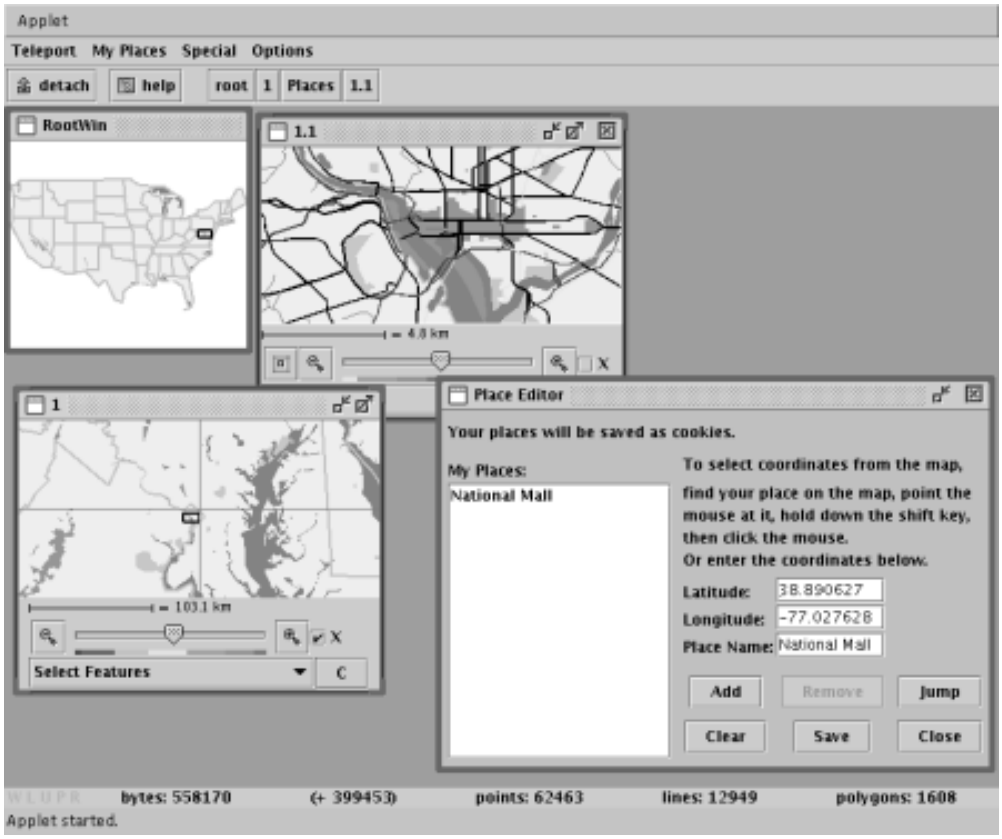

Figure 4.4: Client with Place Editor

- **P** is not currently being used; it never lights up. It may be eventually removed from the interface.
- **R** indicates that a feature is currently being rendered to the screen.

Because the client is multi-threaded, it is normal for more than one letter to be lit at once. The **W** light is especially useful, as it clears up any potential uncertainty about whether or not a window is showing all the information available at the current level-of-detail, as opposed to waiting for more data from the server. The lights also provide a crude form of testing. For example, in the current version the **R** light tends to stay on for a significant amount of time after the others have turned off, which suggests that the rendering is a performance bottleneck.

Chapter 5

Conclusions and Future Work

The feasibility of responsive visualization of a large dataset over a thinwire is demonstrated by this research. We have developed a multi-threaded client-server architecture for responsive visualization in a thinwire environment. In such an environment, the network bandwidth between the user and the data (in this case, between the client and the server) is highly volatile and unpredictable. But even if bandwidth temporarily drops to zero, the system continues to respond to user requests intelligently and promptly, using whatever information it has available. The architecture is specifically designed for responsiveness. We have developed a sophisticated form of priority queue, used throughout the system, that is specially designed to match task priorities to user interest, and the system architecture allows the propagation of priorities in order to quickly reflect changes in user behavior.

We have developed algorithms and tools for interpreting the TIGER dataset and unifying the county-based data into a single dataset—a difficult process that requires handling many anomalies and messy "real world" problems. We have devised a mostly automatic approach to map generalization that generates reasonable, though not excellent, maps. And we have preprocessed the data into a highly efficient database schema.

Our system uses a flexible and scalable visualization interface that uses multiple linked windows. It allows oriented viewing of a dataset that has a zoom factor, from highest overview to highest detail, of almost 10,000. As far as we know, this is the first web-based visualization system with smooth zooming and panning for the entire continental United States, or for any equivalently large geographic dataset.

In many ways, however, this research is really just beginning. Much of the

work described in this dissertation has been focused on simply producing a system that works at all. Now that we have a system, we can start to look at improving it, extending it, and applying it to different situations. The remainder of this chapter covers the many directions that we see where this research can go.

5.1 Performance Improvements

As was discussed in section 2.5.2, there are several difficulties in designing meaningful testing methods for this system. The development of sophisticated testing procedures for systems like this one would be extremely beneficial for evaluating system designs. We are also interested in formal modeling, perhaps using the statechart method or Harel, et al. [38, 39, 40], which might allow a more rigorous investigation into the responsiveness of the system.

If the difficulties with testing can be overcome, there are a number of changes that could be made to the system that might improve performance, and whose value could be determined by testing. Some of these possible changes are listed below. It is unlikely that any one of these will have a dramatic effect, but a combination of small improvements could be quite significant.

1. On the server side, when the Data Provider gets a new request for a blob that has already been sent to the Network Writer, it upgrades the blob's priority in the Network Writers input queue, if the blob is still there. If the blob has already been sent to the client, nothing happens, but perhaps something should. The server could send an *upgrade blob* message to the client, and if the blob is still on the unpacker's input queue, it can be upgraded.

2. The current responsibilities of the Event Handler might be split into two threads. The Event Handler would still handle the immediate response to mouse events, such as translating the screen image, but the responsibility of placing tasks onto the Network Writer queue and the Searcher queue could be passed off to another thread, thus freeing up the Event Handler to handle mouse events that much more quickly.

3. We should be able to find more efficiency in the rendering process. In the current system, the same geographic object might be rendered multiple times unnecessarily, if the object is contained in more than one package that is concurrently on the Renderer priority queue. An alternative would be to use an upgradeable priority queue, similar to the Response Sender queue,

and store individual objects on the queue directly, instead of packages. We might have a function `upgradeOrAdd` that upgrades the priority of an object if it is already on the queue, and adds it to the queue if it is not. To preserve the correct ordering of polygons, polylines and landmarks, the polygons would be given priority of $requestNumber * 4$, polylines $requestNumber * 4-1$, landmarks $requestNumber*4-2$, and commit $requestNumber*4-3$. There is a problem with this idea that would need to be solved, though: the whole object should be drawn, not just the portion that intersects the bounding box of the request that it is in response to. Now suppose that one corner of a polygon intersects a request, causing the whole polygon to be redrawn. But a line that lies within another part of the polygon that is also on the screen would not be considered part of the request, and therefore would not be redrawn, so the new rendering of the polygon will overdraw the line.

4. Use prefetching (see [36, 46, 47], for example). Some prefetching is already implicit in the system by virtue of the fact that the bounding boxes of the blobs that are stored on the server will frequently extend outside the range that is requested, and so the extra data in such a blob is de facto prefetched. The simplest way to increase prefetching is to explicitly request data for ranges that are larger than the current window. For the purposes of constructing requests to send to the server, we might define a logical window that is twice the width and height of the actual window, and always centered on the same point. A more sophisticated system might predict future requests based on the recent past: if we have been panning in a certain direction, assume we will be panning in that direction in the future.

5. Our current (online) scheduling approach is essentially that request for the window with current input focus have highest priority, and among requests for the same window the most recent request has highest priority. Threads execute tasks greedily based on the current state of local information. Other, more sophisticated approaches, are possible. For example, a long task might be delayed in the hopes that a more important shorter task will be soon coming along. See [11] for a discussion of online scheduling based on a similar thinwire visualization problem for large images.

6. If the server could recognize that the client has all the data needed for a request before it makes the query to the database, we might be able to save disk accesses on the server side. For example, the server might be

able to store the union of the bounding boxes of all previous requests, and if the difference of the new request from that union is the empty set, the query is unnecessary. On the other hand, since the DBMS keeps its own in-memory cache, it could be that the DBMS is essentially already doing this for us, since queries that duplicate the work of previous requests could be answered from the DBMS cache, without any disk accesses. So doing it ourselves would be unnecessary duplication of effort.

7. When panning, there is a tendency to put the main area of interest to the middle of the window. But this means that the last few pans may have resulted in requests for data at the edge of the window, and these would have highest priority since they would be the most recent. So maybe tasks near the current center of the window should have highest priority, instead of those for the most recent request.

8. While we currently only use one TCP connection between the server and the client, it is possible that performance would improve with two or more connections, by more fully using the available thinwire bandwidth. The change would be relatively simple to make. Referring to figure 2.1, two (or more) network writer threads on the server side would read blobs from the output queue, and each would connect to a separate network reader thread on the client side. These multiple network readers would each write to a single unpacker queue.

9. Much of the client side computation involving location coordinates and ranges is currently done using double precision floating point arithmetic, but a back of the envelope calculation shows that at our lowest scale (highest detail), 2.5 meters per pixel, the number of pixels required to stretch across the country is small enough to fit into a 32 bit integer. So we could store the coordinates as integers, which would save memory and probably improve performance as well.

5.2 Visualization Interface

There are a number of improvements that can be made in usability and functionality. The labeling can clearly be improved, by using "excentric" labels [23] or label sampling [10], or perhaps by including a limited number of static labels with the dynamic pop-up labels. We have begun to implement a rudimentary feature search capability—a small first step toward enhancing the client to allow

the kind of analysis and planning that is found in GIS. Eventually we would like to let the user make queries, such as "What is the population of this area?", "Where are the nearest schools?" or "What is the address range on this block?" This will require storing more descriptive information in the database. Once we have a good way to store and retrieve address information, we can start to think about route planning and, ideally, route visualization. The space-scale diagrams of Furnas and Bederson [35] can be used for visualizing pan-zoom trajectories. If a user wants to see the route from an address in one city to an address in another, an animation might start at a high detail level and show the path from the start address to the nearest highway, then zoom out and follow the highway to the destination city, then zoom in and follow the route along city streets to the destination address.

5.3 Other Versions

This section looks at what changes would have to be made to apply the system to different application domains or in different implementation environments. Section 5.3.1 looks at using HTTP tunneling so the client can be run from behind firewalls; section 5.3.2 looks at making the client work on cell phones and PDAs; and section 5.3.3 looks at abstracting out the core of the client-server communication and turning this into a generic package that could be used by other applications.

5.3.1 HTTP Tunneling

Most firewalls do not allow direct socket connections, which is how we are currently doing the client-server communication. Many firewalls only allow HTTP requests and responses, and some only on port 80. So if we want the program to be usable by people behind firewalls (which includes many, if not most, people at work) then one solution is HTTP tunneling. The idea is fairly simple: all requests are packaged as HTTP requests, and all responses are packaged as HTTP responses. On the server side, the client connection would be handled by a web server, and the map server would be implemented within the web server using Java servlets, CGI program, or similar technology. To make a range request, the client would simply open a URL, and include the range as a request parameter. Either a GET or a POST request can be used. The request is handled by a servlet or CGI program, which queries the database and sends the proper data back as an HTTP response.

But there are complications. The main problem is that in our system the requests are not responded to in the same order that they are made. This needs to be made to work under either HTTP/1.0 or HTTP/1.1 (with pipelining). Under HTTP/1.0, each HTTP request requires a new connection. From the client's perspective, reading a URL is a synchronous operation, so if we want multiple requests to be serviced concurrently, the Request Sender will need to spawn a new thread for each URL. This would be a simple, short-lived thread that just stores the HTTP response in a byte array, puts that byte array (blob) onto the Unpacker's priority queue, and exits.

On the server side, a new short-lived servlet thread would be spawned to handle each HTTP request. Each servlet thread would put its own task onto the Data Provider priority queue, and then go to sleep. When the Response Sender decides that it is time for that thread to send a blob in response, it would wake the thread up. Since there can be multiple blobs in each response, the servlet thread would go back to sleep after sending each blob, and the Response Sender would be responsible for telling it when it can finally exit.

This approach might introduce a new problem for performance by starting many concurrent threads on both the client and server. More investigation will need to be done to see if this is a problem, and if so, whether a way can be found to limit the number of threads without losing the prioritization of responses.

Under HTTP/1.1, multiple requests can be pipelined onto a single connection, but the HTTP/1.1 specification says that the requests are responded to in the same order that they are sent in. To preserve our responsive prioritization of requests, the server could send whichever blob is highest priority to whichever request is next in order; on the client side, all blobs go onto the Unpacker priority queue, so the client doesn't need to match up a blob with the request that generated it.

5.3.2 Cell Phones and PDAs

A version for cell phones and PDAs will require major changes to the visualization interface, and to other aspects of the system also. The multiple window capabilities will have to be abandoned; the interface reduced to a single window with zoom and replace, and only the most necessary controls. Probably the client will have to be revised to use integer arithmetic only (J2ME, the Java "micro edition", provides no floating point capabilities). And more generally reducing the client side memory footprint will have to become a primary design objective. A minimum cache size will have to be set, and the cache pruning policy redesigned.

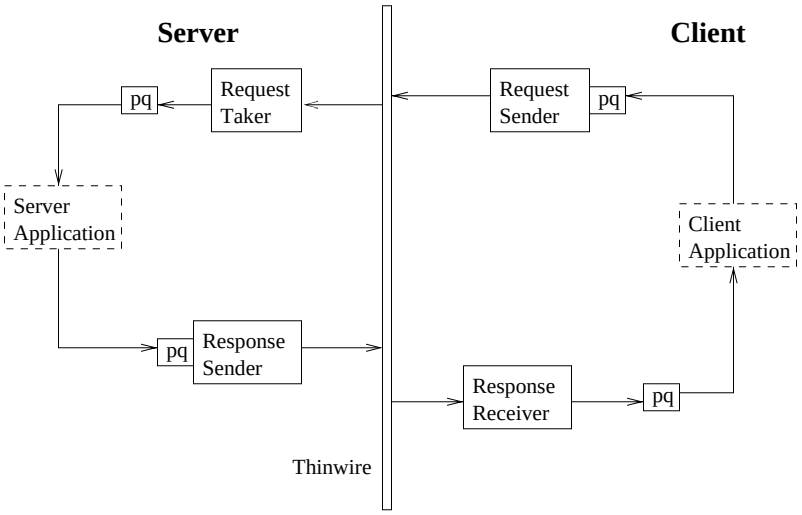

Figure 5.1: Client-Server Kernel

(Currently there is no hard limit on the size of the cache; pruning is triggered by a certain amount of data being downloaded, and is not guaranteed to remove any data at all.) The preprocessor will have to store smaller blobs in the database, which will probably have a negative impact on performance.

5.3.3 Responsive Client-Server Kernel

It seems that the core of our client-server communication and task processing can be abstracted out and used in other application domains, not just geographic visualization. We could provide a kind of client-server "kernel", which would provide a programming interface for client and server applications. The interface would include methods for sending tasks, receiving tasks, changing the priority of tasks, etc. It would leave generating and handling the tasks to the application programs. This is similar in spirit to the Petra-Flow framework of Forman [26], although Petra-Flow is only a client-side architecture. Figure 5.1 shows a possible client-server kernel architecture.

5.4 Dataset

Changes in our approach to data processing and storage might lead to improvements in performance and a smoother visualization experience. A relatively sim-

ple and straightforward change would be to index the blobs in the database using four byte integer rather than eight byte double precision coordinates. The raw data gives latitude and longitude coordinates to six decimal places, so there are never more than nine significant digits, which can fit in a four byte integer. This change would reduce the size of the indexes by about half, doubling the number of index tuples that can fit on a page, which should improve performance. But R-tree indexing of integer coordinate ranges is not built in to PostgreSQL, so we would have to implement that feature, either by extending PostgreSQL's R-tree, or by using its Generalized Search Tree (GiST) [42].

The generalization process can certainly be improved, possibly by incorporating a limited amount of manual labor into the computation of LODs. We did this in an early stage of the project, but abandoned it for fully automated generalization because of the amount of labor needed.

A much more radical change would be to accommodate fine-grained, virtually continuous changes in detail, rather than discrete S-Levels. The topological information might be handled by data structures like progressive meshes and others [44, 79, 66], and the geographical information can be handled with van Oosterom's reactive data structures (see section 3.1). All of this will introduce a serious performance reduction, so making a system with these changes that is fast enough will be a challenge.

Once fine-grained changes in detail are implemented, we can start to think about foveation of the data/image [12, 13], or fisheye views (see section 4.1.1), in which different parts of a window or image show data at different resolutions or LODs. This might decrease the bandwidth required and increase responsiveness, although with the high semantic content of geographic data, it's not clear how to smoothly display multiple detail levels in the same image.

Appendix

Singleton Constraints:

These constraints are caused by one county being completely contained within another, so that the two must be grouped together.

1. Winchester, VA 51840 with Frederick, VA 51069
2. Waynesbor, VA 51820 with Augusta, VA 51015
3. Staunton, VA 51790 with Augusta, VA 51015
4. Norton, VA 51720 with Wise, VA 51195
5. Martinsville, VA 51690 with Henry, VA 51089
6. Lexington, VA 51678 with Rockbridge, VA 51163
7. Harrisonburg, VA 51660 with Rockingham, VA 51165
8. Fairfax, VA 51600 with Fairfax, VA 51059
9. Emporia, VA 51595 with Greensville, VA 51081
10. Covington, VA 51580 with Alleghany, VA 51005
11. Clifton Forge, VA 51560 with Alleghany, VA 51005
12. Charlottesville, VA 51540 with Albemarle, VA 51003
13. Buena Vista, VA 51530 with Rockbridge, VA 51163
14. Bedford, VA 51515 with Bedford, VA 51019
15. Nantucket, MA 25019 with Dukes, MA 25007

Multiple Piece Constraints:

These constraints deal with situations where a county is composed of multiple pieces, and because of the way that the multiple pieces are situated in and around surrounding counties, two or more counties must be grouped together.

1. Fulton, KY 21075 with New Madrid, MO 29143
2. Dickson TN 47043 with Cheatham TN 47021
3. Loudon TN 47105 with McMinn TN 47107, Monroe TN 47123, and Roane TN 47145
4. Tipton TN 47167 with Mississippi AR 05093
5. White TN 47185 with Van Buren TN 47175, Bledsoe TN 47007, or Cumberland TN 47035 (not sure which)
6. Jackson AL 01071 with Marshall AL 01095
7. Brooks GA 13027 with Lowndes GA 13185
8. Lee GA 13177 with Terrell GA 13273
9. Macon GA 13193 with Taylor GA 13269
10. Taylor GA 13269 with Schley GA 13249
11. Upson GA 13293 with Pike GA 13231
12. Hoke NC 37093 with Scotland NC 37165
13. Pickens SC 45077 with Oconee SC 45073
14. Dixie FL 12029 with Gilchrist FL 12041
15. Monroe FL 12087 has an island in the ocean, not connected
16. Fairfax VA 51059 with 51600, which is within 51059
17. Montgomery VA 51121 with Radford VA 51750
18. Prince George VA 51149 with Hopewell VA 51670
19. Prince William VA 51153 with Manassas VA 51683

20. Bristol VA 51520 with Washington VA 51191
21. Manassas Park VA 51685 with Prince William VA 51153
22. Chester PA 42029 with Delaware PA 42045
23. New York NY 36061 with Hudson NJ 34017
24. Washington RI 44009 has Block Island out in the ocean, not connected
25. Norfolk MA 25021 with Plymouth MA 25023 and Suffolk MA 25025
26. Caledonia VT 50005 with Grafton NH 33009
27. Los Angeles CA 06037 has two islands in the ocean, not connected
28. Santa Barbara CA 06083 has two islands in the ocean, one not connected, one connected to a Ventura piece (see below)
29. Ventura CA 06111 has two islands in the ocean, one not connected, one connected to a Santa Barbara piece (see below)
30. Santa Barbara CA 06083 with Ventura CA 06111 (one piece from each in ocean but connected with each other)
31. San Francisco CA 06075 has an island in the ocean, not connected
32. Adams CO 08001 with Denver CO 08031 (Adams has two pieces in Denver)
33. Denver CO 08031 with Arapaho CO 08005 (Denver has one piece in Arapaho; Arapaho has ten pieces in Denver)
34. Jefferson CO 08059 with Denver CO 08031 (Jefferson has three pieces in Denver)
35. Jefferson CO 08059 with Denver CO 08031 and Arapaho CO 08005 (Jefferson has two pieces between Denver and Arapaho)
36. Sandoval NM 35043 with Los Alamos NM 35028 (Sandoval has a piece between Los Alamos and Santa Fe 35049)
37. Lac qui Parle MN 27073 with Big Stone MN 27011 (Lac qui Parle has a piece between Big Stone and Grant SD 46051 (or maybe Roberts SD 46109))

38. Madison LA 22065 with Warren MS 28149 (Madison has a piece between Warren and Tensas LA 22107)

39. St. Bernard LA 22087 has islands in the ocean, not connected (one piece)

40. St. Martin LA 22099 with Iberia LA 22045 (St. Martin has two pieces, with Iberia in between them)

41. West Feliciana LA 22125 with Pointe Coupee LA 22077 (West Feliciana has a piece touching Pointe Coupee, Concordia LA 22029 and Avoyelles LA 22009)

Bibliography

[1] Patrick Baudisch, Nathaniel Good, Victoria Belloti, and Pamela Schraedley. Keeping things in context: A comparative evaluation of focus plus context screens, overviews, and zooming. In *CHI 2002*. ACM Press, 2002.

[2] Patrick Baudisch, Nathaniel Good, and Paul Stewart. Focus plus context screens: Combining display technology with visualization techniques. In *UIST 2001*. ACM Press, 2001.

[3] Benjamin B. Bederson and James D. Hollan. Pad++: A zooming graphics interface for exploring alternate interface physics. In *Proceedings UIST '94*, pages 17–26, 1994.

[4] Benjamin B. Bederson, James D. Hollan, Ken Perlin, Jonathan Meyer, David Bacon, and George Furnas. Pad++: A zoomable graphical sketchpad for exploring alternate interface physics. *Journal of Visual Languages and Computing*, 7:3–31, 1996.

[5] Benjamin B. Bederson, Larry Stead, and James D. Hollan. Pad++: Advances in multiscale interfaces. In *ACM SIGCHI '94*. ACM Press, 1994.

[6] T. Berners-Lee, R. Fielding, and H. Frystyk. HTTP/1.0 - informational RFC 1945, May 1996. `http://www.w3c.org/Protocols/rfc1945/rfc1945`.

[7] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[8] B. Shneiderman C. Ahlberg. Visual information seeking: Tight coupling of dynamic query filters with starfield displays. In *Proceedings of ACM CHI94 Conference*, pages 313–317, 1994.

[9] Stuart K. Card, Jock D. Mackinlay, and Ben Shneiderman. *Readings in Information Visualization: Using Vision to Think.* Morgan Kaufmann, 1999.

[10] Matthew Chalmers, Robert Ingram, and Christoph Pfranger. Adding imageability features to information displays. In *Proc. ACM Symposium on User Interface Software and Technology (UIST96)*, pages 33–39, 1996.

[11] Ee-Chien Chang and Chee Yap. Competitive online scheduling with level of service. *Journal of Scheduling*, October 2000. Special Issue on Online Algorithms. Also: in Proc. 7th Ann. Intl. Computing and Combinatorics Conf. (COCOON), August 20-23, 2001, Guilin, China.

[12] Ee-Chien Chang, Chee Yap, and Ting-jen Yen. Realtime visualization of large images over a thinwire. In *IEEE Visualization '97 (Late Breaking Hot Topics)*, pages 45–48, 1997. See also CD proceedings of conference.

[13] Ee-Chien Chang and Chee K. Yap. A wavelet approach to foveating images. *ACM Symp. on Computational Geometry*, 13:397–399, 1997.

[14] T.-W. Chen, P. Krzyzanowski, M. R. Lyu, C. Sreenan, and J. A. Trotter. Renegotiable quality of service—a new scheme for fault tolerance in wireless networks. In *Proceedings FTCS-27*, pages 21–30, 1997.

[15] Jimmy H. P. Chim, Rynson W. H. Lau, Antonio Si, Hong Va Leong, Danny To, Mark Green, and Miu Ling Lam. Multi-resolution model transmission in distributed virtual environments. In *Proceedings of ACM Symposium on Virtual Reality Software and Technology*, pages 25–34, November 1998.

[16] Jon Christensen, Joe Marks, and Stuart Shieber. An empirical study of algorithms for point-feature label placement. *Transactions on Graphics*, 14(3), July 1995.

[17] William Cleveland. *Visualizing Data.* Hobart Press, 1993.

[18] Daneil Cohen-Or. Model-based view-extrapolation for interactive VR websystems. In *Computer Graphics International '97*, pages 104–112, 1997.

[19] Daniel Cohen-Or and Eyal Zadicario. Visibility streaming for network-based walkthroughs. In *Graphics Interface*, pages 1–7, June 1998.

[20] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[21] MapQuest Corporation. Map server web site. www.mapquest.com.

[22] D. H. Douglas and T. K. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Canadian Cartographer*, 10(2):112–122, 1973.

[23] Jean-Daniel Fekete and Catherine Plaisant. Excentric labeling: Dynamic neighborhood labeling for data visualization. In *Proceedings of CHI '99*, pages 512–519, 1999.

[24] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol—HTTP/1.1 draft standard RFC 2616, June 1999. `ftp://ftp.isi.edu/in-notes/rfc2616.ps`.

[25] R. Finkel and J. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica*, 4:1–9, 1974.

[26] George Forman. *Obtaining Responsiveness in Resource-Variable Environments*. PhD thesis, University of Washington, 1996.

[27] George Forman. Wanted: Programming support for ensuring responsiveness despite resource variability and volatility. Technical report, Hewlett-Packard Labs, 1998.

[28] George H. Forman and John Zahorjan. The challenges of mobile computing. *IEEE Computer*, 27(4):38–47, April 1994. Also in *Mobility: Processes, Computers and Agents*, Addison-Wesley, 1999.

[29] Thomas A. Funkhouser. *Database and Display Algorithms for Interactive Visualization of Architectural Models*. PhD thesis, UC Berkeley, 1993.

[30] Thomas A. Funkhouser. Database management for interactive display of large architectural models. In *Graphics Interface*, pages 1–8, 1996.

[31] Thomas A. Funkhouser and Carlo H. Séquin. Adaptive display algorithm for interactive frame rages during visualization of complex virtual environments. In *Computer Graphics Proceedings*, pages 247–254, 1993.

[32] Thomas A. Funkhouser, Carlo H. Séquin, and Seth J. Teller. Management of large amounts of data in interactive building walkthroughs. In *Computer Graphics (1992 SIGGRAPH Symposium on Interactive 3D Graphics)*, pages 11–20, 1992.

[33] Thomas A. Funkhouser, Seth J. Teller, Carlo H. Séquin, and Delnaz Khorramabadi. The UC Berkeley system for interactive visualization of large architectural models. *Presence*, 5(1), January 1996.

[34] George W. Furnas. Generalized fisheye views. In *Human Factors in Computing Systems CHI '86 Conference Proceedings*, pages 16–23, 1986.

[35] George W. Furnas and Benjamin B. Bederson. Space-scale diagrams: Understanding multiscale interfaces. In *CHI '95 Proceedings*. ACM Press, 1995.

[36] Carsten A. Gerlhof and Alfons Kemper. A multi-threaded architecture for prefetching in object bases. *Lecture Notes in Computer Science*, 779:351–364, 1994.

[37] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In Beatrice Yormark, editor, *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts , June 18-21, 1984*, pages 47–57. ACM Press, 1984.

[38] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[39] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. In *IEEE Transactions on Software Engineering*, volume 16, pages 403–414, 1990.

[40] David Harel and Michal Politi. *Modeling Reactive Systems with Statecharts: The Statemate Approach*. McGraw-Hill, 1998.

[41] Paul S. Heckbert and Michael Garland. Multiresolution modeling for fast rendering. In *Graphics Interface '94*, pages 43–50, 1994.

[42] Joseph M. Hellerstein, Jeffery F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In *Proceedings 21st International Conference on Very Large Data Bases*, pages 562–573, 1995.

[43] Gerd Hesina and Dieter Schmalstieg. A network architecture for remote rendering. In *Proceedings of 2nd International Workshop on Distributed Interactive Simulation and Real Time Applicatioins (DIS-RT '98)*, pages 88–91, 1998.

[44] Hugues Hoppe. Progressive meshes. In *Computer Graphics (ACM SIGGRAPH 1996 Proceedings)*, pages 99–108, 1996.

[45] Brian Kernighan and Christopher Van Wyk. Extracting geometric information from architectural drawings. In *WACG: 1st Workshop on Applied Computational Geometry: Towards Geometric Engineering, WACG*. LNCS, 1996.

[46] Nils Knafla. Speed up your database client with adaptable multithreaded prefetching. In *Proceedings of the Sixth IEEE International Symposium on High Performance Distributed Computing*, pages 102–111, 1997.

[47] Nils Knafla. An adaptable multithreaded prefetching technique for client-server ojbect bases. *Cluster Computing*, 1(1):27–37, 1998.

[48] Marcel Kornacker and Douglas Banks. High-concurrency locking in r-trees. In *The VLDB Journal*, pages 134–145, 1995.

[49] J. Lamping and R. Rao. The hyperbolic browser: A focus + context technique for visualizing large hierarchies. *Journal of Visual Languages and Computing*, 7(1):33–55, 1996.

[50] John Lamping, Ramama Rao, and Peter Pirolli. A focus+context technique based on hyperbolic geometry for visualizing large hierarchies. In *CHI '95 Proceedings*. ACM Press, 1995.

[51] Y. K. Leung and M. D. Apperley. A review and taxonomy of distortion-oriented presentation techniques. *ACM Transactions on Computer-Human Interaction*, 1(2):126–160, 1994.

[52] Alan M. MacEachren. An evolving cognitive-semiotic approach to geographic visualization and knowledge construction. *Information Design Journal*, 10(1):26–36, 2001. Special issue on Jacques Bertin's theories.

[53] Alan M. MacEachren, Robert Edsall, Daniel Haug, Ryan Baxter, George Otto, Raymon Masters, Sven Fuhrmann, and Liujian Qian. Virtual environments for geographic visualization: Potential and challenges. In *Proceedings of the ACM Workshop on New Paradigms in Information Visualization and Manipulation*, pages 35–40, 1999.

[54] Yair Mann and Daniel Cohen-Or. Selective pixel transmission for navigating in remote virtual environments. *Computer Graphics Forum*, 16(3):201–206, September 1997.

[55] André Matos, Jonas Gomes, and Luiz Velho. Cache management for real time visualization of 2d data sets. In *SIBGRAPI '98*, 1998.

[56] B.H. McCormick, T.A. DeFanti, and M.D. Brown. Visualization in scientific computing. *ACM Computer Graphics (special issue)*, 21(6), 1987.

[57] Kurt Mehlhorn. *Datastructures and Algorithms 3: Multi-Dimensional Searching and Computational Geometry*. Springer-Verlag, 1984.

[58] David A. Nation, Catherine Plaisant, Gary Marchionini, and Anita Komlodi. Visualizing websites using a hierarchical table of contents browser: WebTOC. In *Proceedings 3rd Conference on Human Factors and the Web*, 1997.

[59] H. Frystyk Nielsen, Mike Spreitzer, Bill Janssen, and Jim Gettys. HTTP-NG overview, INTERNET-DRAFT, November 1998. work in progress.

[60] Henrik Frystyk Nielsen, James Gettys, Anselm Baird-Smith, Eric Prud'hommeaux, Håkon Wium Lie, and Chris Lilley. Network performance effects of HTTP/1.1, CSS1, and PNG. *Computer Communication Review*, 27(4), October 1997.

[61] NYU Active Visualization Project. Map server demo. cs.nyu.edu/visual/home/demos/tigerDemo/.

[62] The Mozilla Organization. HTTP/1.1 pipelining faq, September 2001. `www.mozilla.org/projects/netlib/http/pipelining-faq.html`.

[63] Renato Pajarola. *Access to Large Scale Terrain and Image Databases in Geoinformation Systems*. PhD thesis, ETH Zürich, 1998. Dissertation No. 12729.

[64] Renato Pajarola. Large scale terrain visualization using the restricted quadtree triangulation. In *Proc. IEEE Visualization '98*, pages 19–26 and 515, 1998.

[65] Renato Pajarola, Thomas Ohler, Peter Stucki, Kornel Szabo, and Peter Widmayer. The alps at your fingertips: Virtual reality and geoinformation systems. In *Proceedings 14th International Conference on Data Engineering, ICDE '98*, pages 550–557, 1998.

[66] Renato Pajarola and Jarek Rossignac. Compressed progressive meshes. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):79–93, January-March 2000.

[67] Renato Pajarola and Peter Widmayer. Virtual geoexploration: Concepts and design choices. *International Journal of Computational Geometry and Applications*, 11(1):1–14, February 2001.

[68] Songju Park, Dongman Lee, Mingyu Lim, and Chansu Yu. Scalable data management using user-gased caching and prefetching in distributed virtual environments. In *VRST 2001*, pages 121–126, 2001.

[69] Ken Perlin and David Fox. Pad: An alternative approach to the computer interface. In *Proceedings 1993 ACM SIGGRAPH*, pages 57–64, 1993.

[70] Catherine Plaisant, David Carr, and Ben Shneiderman. Image browsers: Taxonomy, guidelines, and informal specifications. *IEEE Software*, 12(2):21–32, 1995.

[71] K. Rothermel, G. Dermler, and W. Fiederer. QoS negotiation and resource reservation for distributed multimedia applications. In *1997 International Conference on Multimedia Computing and Systems (ICMCS '97)*, 1997.

[72] Szymon Rusinkiewicz and Marc Levoy. Streaming QSplat: a viewer for networked visualization of large, dense models. In *Symposium on Interactive 3D Graphics*, pages 63–68, 2001.

[73] Hanan Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Addison-Wesley, 1990.

[74] Hanan Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1990.

[75] Doug Schaffer, Zhengping Zuo, Saul Greenberg, Lyn Bartram, John Dill, Shelli Dubs, and Mark Roseman. Navigating hierarchically clustered networks through fisheye and full-zoom methods. *ACM Transactions on Computer-Human Interaction*, 3(2):162–188, June 1996.

[76] V. F. Schenkelaars. Implementation of reactive data structures for postgres. Master's thesis, FEL-TNO Divisie 2, 1992. Tech. Rep. FEL-92-S343 (not found yet).

[77] Dieter Schmalstieg. *The Remote Rendering Pipeline: Managing Geometry and Bandwidth in Distributed Virtual Environments*. PhD thesis, Vienna University of Technology, 1997.

[78] Dieter Schmalstieg and Michael Gervautz. Demand-driven geometry transmission for distributed virtual environment. *Computer Graphics Forum*, 15(3):421–432, 1996.

[79] Dieter Schmalstieg and Gernot Schaufler. Smooth levels of detail. In *Proceedings of IEEE Virtual Reality Annual International Symposium (VRAIS '97)*, pages 12–19, 1997.

[80] Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley, third edition, 1998.

[81] Lidan Shou, Jason Chionh, Zhiyong Huang, Yixin Ruan, and Kian-Lee Tan. Walking through a very large virtual environment in real-time. In *Proceedings of the 27th VLDB Conference*, 2001.

[82] R. Spence and M. D. Apperley. Data base navigation: An office environment for the professional. *Behavior and Information Technology*, 1(1):43–54, 1982.

[83] Simon Spero. Next generation hypertext transport protocol, March 1995. work in progress.

[84] Simon E. Spero. Analysis of HTTP performance problems, July 1994. `http://www.w3.org/Protocols/HTTP/1.0/HTTPPerformance.html`.

[85] Erik B. Steiner, Alan M. MacEachren, and Diansheng Guo. Developing and assessing light-weight data-driven exploratory geovisualizatoin tools for the web. In *20th International Cartographic Conference Proceedings*, 2001.

[86] sven Fuhrmann and Alan M. MacEachren. Navigation in desktop geovirtual environments: Usability assessment. In *20th International Cartographic Conference Proceedings*, 2001.

[87] Kornel Szabo, Peter Stucki, Patrick Aschwanden, Thomas Ohler, Renato Pajarola, and Peter Widmayer. A virtual reality based system environment for intuitive walk-throughs and exploration of large-scale tourist information. In *Proc. of the Enter95 Conference*, pages 10–15, 1995.

[88] Eyal Teler and Dani Lischinski. Streaming of complex 3D scenes for remote walkthroughs. *Computer Graphics Forum*, 20(3), 2001.

[89] Seth J. Teller and Carlo H. Séquin. Visibility preprocessing for interactive walkthroughs. *Computer Graphics*, 25(4):61–69, July 1991. Proceedings of SIGGRAPH'91.

[90] Lisa Tweedie. Characterizing interactive externalizations. In *CHI '97*, 1997. Also in *Readings in Information Visualization: Using Vision to Think*, Card, et al., editors, Morgan Kaufmann, 1999.

[91] U.S. Census Bureau, Washington, DC. *Census 2000 TIGER/Line Files Technical Documentation*, 2000. www.census.gov.

[92] DC U.S. Census Bureau Washington, 2000. Census 2000 TIGER/Line Files [machine-readable data files].

[93] P. van Oosterom and T. Vijlbrief. The spatial location code. In *Proceedings of International Symposium on Spatial Data Handling*, 1996.

[94] Peter van Oosterom. A reactive data structure for geographic information systems. In *Auto-Carto 9 Proceedings*, pages 665–674, 1989.

[95] Peter van Oosterom. *Reactive Data Structures for Geographic Information Systems*. PhD thesis, Leiden University, 1990.

[96] Peter van Oosterom. The reactive-tree: A storage structure for a seamless, scaleless geographic database. In *Proceedings Auto-Carto 10*, pages 393–407, 1991.

[97] Peter van Oosterom. The GAP-tree: An approach to 'on-the-fly' map generalization of an area partitioning. In J. C. Mueller, J. P. Lagrange, and R. Weibel, editors, *GIS and Generalization: Methodology and Practice*, chapter 9, pages 120–132. Tayler & Francis, London, 1995.

[98] Peter van Oosterom and Christiaan H. J. Lemmen. Efficient access to a very large spatial database. In *Joint European Conference and Exhibition on Geographical Information, JEC96*, 1996.

[99] Peter van Oosterom and Chrit Lemmen. Spatial data-management on a very large cadastral database. *Computers, Environment and Urban Systems*, 25(4-5):509–528, 2001.

[100] Peter van Oosterom and Vincent Schenkelaars. The development of an interactive multi-scale GIS. *International Journal of Geographical Information Systems*, 9(5):489–507, 1995.

[101] Petrus van Oosterom. *Reactive Data Structures for Geographic Information Systems*. Oxford University Press, 1993.

[102] Tom Vijlbrief and Peter van Oosterom. The GEO++ system: An extensible GIS. In *Proceedings 5th International Symposium on Spatial Data Handling*, pages 40–50, 1992.

[103] W3C. Final HTTP-NG activity statement, 2001. `http://www.w3.org/Protocols/HTTP-NG/Activity.html`.

[104] Robert Weibel. Generalization of spatial data. Course Notes for the CISM Advanced School on Algorithmic Foundations of Geographical Information Systems, September 1996.

[105] Chee Yap, Kenneth Been, and Zilin Du. Responsive thinwire visualization: Application to large geographic datasets. In *Proc. SPIE Vol. 4665, Visualization and Data Analysis 2002*, pages 1–12. 2002.

[106] Chee Yap and Ting-jen Yen. Design and instrumentation of a thinwire visualization system, 2000. unpublished manuscript.

[107] Christopher Zach and Konrad Karner. Prefetching policies for remote walkthroughs. In *WSCG 2002*, 2002.