A Responsive Architecture for Thinwire Visualization*

Kenneth Been[†] and Chee Yap
Department of Computer Science
Courant Institute, New York University
251 Mercer Street
New York, NY 10012
been,yap@cs.nyu.edu

November 9, 2002

Abstract

We describe a multi-threaded client-server architecture for thinwire visualization of large datasets. A thinwire refers to a network connection in which the bandwidth is not only small but also volatile. In contrast to traditional approaches used in walkthroughs and geographic visualization, we do not design with the goal of maintaining a given frame rate—a paradigm that is difficult to maintain in a thinwire environment. Instead we focus on responsiveness, and use some properties of human conversation to suggest system behaviors that maximize perceived responsiveness. The system extensively uses concurrency, along with sophisticated data structures for task reordering and cancellation, to bring about these behaviors. We verify the correctness of our cache management protocols; correctness is non-trivial because of the task reordering and cancellation. We have implemented these techniques in a web-based interactive visualizer for a geographic description of the entire continental United States, down to street level.

Index Terms—Visualization, thinwire, responsiveness, client-server, multi-threading, Geographical Information Systems (GIS), visualization architecture, cache consistency.

1 Introduction

The maturation of the desktop computer over the last few years has cleared the path for previously high-end visualization techniques to be used in mass-market products [4]. In turn, the near ubiquity of the web has inspired work on web-based visualization [2], in which the (large) dataset resides in a remote location and is visualized through a browser plugin or Java applet. One common application that is ripe for responsive, interactive, web-based visualization is the browsing of road maps. This is the reference application for the responsive architecture presented in this paper.

We model the Web with the notion of a *thinwire*, which represents bandwidth that is not only low but also unpredictable—it might drop to zero temporarily, if the network becomes disconnected. Forman [11, 12] describes such a bandwidth as "variable and volatile", and with

*This research has been supported in part by NSF grants #CCR-9619846 and #CCR-0082056
[†]corresponding author

the growth of wireless networks, bandwidth is as volatile as ever [13]. In such an environment, we cannot assume a minimum bandwidth. Geographic visualization systems such as the ViRGIS [32, 33, 31, 37] and GeoVISTA [27, 36, 28, 14] projects give little or no attention to thinwire issues, but we can gain insight into the problem by looking at similar problems, especially architectural walkthroughs.

In the traditional walkthrough, as exemplified by Funkhouser, et al. [18, 17, 15, 19, 16], the guiding paradigm is the model of a sequence of frames, taken from film and video. In this model we typically have a rendering pipeline with stages for I/O, visibility determination, rendering, and possibly others, along with a feedback loop for user navigation. Funkhouser [15] explicitly targets a frame rate of ten frames per second. Schmalstieg, et al., [35, 34, 24], extended the rendering pipeline across the network to produce the Remote Rendering Pipeline, and introduced techniques to reduce the bandwidth required in order to maintain the frame rate. Mann and Cohen-Or [29] used graphics techniques to have the client extrapolate what its next few frames will be using a "textureless model", so that the server generally only sends corrections, rather than complete frames. Cohen-Or and Zadicario [9] use a streaming server that sends a superset of the data requested, so that one message transmission can suffice for multiple frames. Teler and Lischinski [38] drop the frames paradigm, at least as far as the server and network are concerned, and instead use prediction by the streaming server to try to make sure that data is at the client before it is needed.

For various reasons, none of the above approaches is sufficient for our requirements. Of course most of them rely on aspects that are specific to walkthrough, such as the nature of the data or the relatively predictable navigation paths. But there is something else more fundamental and interesting: each of them is guided by a paradigm which says that whenever the user makes any kind of request, the system's response should be given immediately (or at least no later than the next frame). For the systems guided by the frames paradigm, this means that there must be an assumption, at least implicitly, that there is some minimum bandwidth separating user from data; this assumption makes those systems inappropriate in a thinwire environment. In Teler and Lischinski's system, it means that the server has the burden of being able to predict what data the user is going to want in the near future. It is our claim that for many visualization applications these requirements are unnecessarily stringent. We take a very different approach, by designing specifically for *responsiveness.* This approach allows us to produce a high quality visual interaction even in a thinwire environment, and without restrictions on, for example, how fast the user can move through the geography. Furthermore, Forman [11] argues that in an environment with volatile resources, designing for responsiveness leads to a more efficient use of resources than either *static sizing* (provide a fixed service level) or *dynamic sizing* (decide at run time what service level can currently be supported).

Since we are dropping the goal of delivering the highest quality image that is possible in time for the next frame, we need a new specification of what information to provide and when. We adopt the paradigm of a human conversation: the behavior and perception of humans in conversation serve as our guide for how a computer should behave in interaction with a human. Now we need to define what we expect in a human conversation.

Card, et al. [4], 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"[4]. 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 suggests some of the behavior of a responsive visualization system. First, whenever the user makes an input action with the mouse, we must ensure that the action is recognized and at least minimally acknowledged within 0.1 seconds, no matter how busy the system is. (In earlier versions of our system we found that even delays in acknowledging mouse drags of about a half second were quite disorienting.) Second, 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 providing it shortly.

Continuing with the human conversation 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, but only if the information is still relevant. Similarly, our responsive system should accept interruptions by always giving highest priority to answering the most recent request, and responses to old requests that have become irrelevant because of newer requests should be dropped completely.

As part of his general programming framework for developing responsive applications in volatile resource environments, Forman [11, 12] identified four principal techniques: feedback before completion, concurrency, dynamic prioritization, and cancellation. In our approach, 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 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. Concurrency is necessary so that one slow task does not hold up all the rest. In this paper we describe a system that makes use of each of these techniques in such a way as to meet the responsiveness goals above. The primary difficulty is in ensuring that the system never gets bogged down by old requests or blocked by low bandwidth.

In the next section we give a brief description of our geographic visualization application. Section 3 covers the multi-threaded architecture for both client and server, and the assignment of tasks to threads. Section 4 describes a sophisticated form of priority queue that we use both for inter-thread communication and for the prioritization of tasks. Section 5 then gives some arguments for the quality of responsiveness that this architecture provides, while some rough testing results are presented in section 6. Section 7 covers our cache management strategies. Ensuring correctness of the cache management turns out to be non-trivial, because of the task reordering and cancellation. Finally, section 8 looks at related work, and some conclusions and ideas for future work are in section 9.

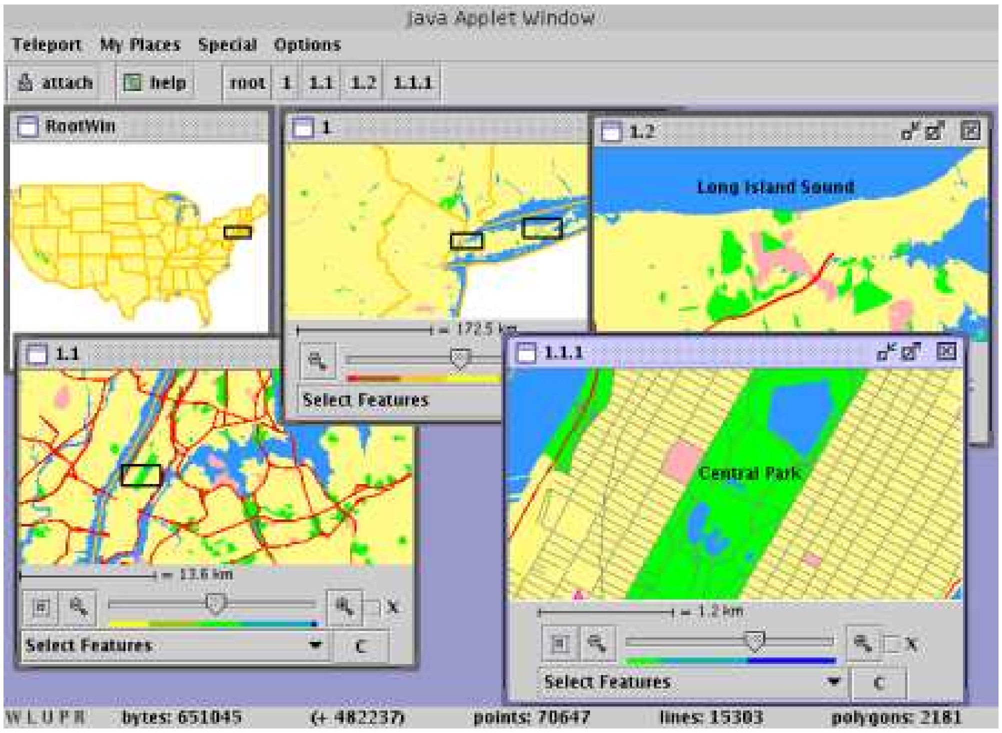

Figure 1: Geographic Visualization Application Screenshot

2 Geographic Visualization Application

We have used the techniques described in this paper to implement a responsive visualization system, with smooth zooming and panning, for the geography of the continental United States, from overview down to street level. Figure 1 shows a screenshot. (By "smooth zooming and panning we mean virtually continuous changes in scale and position, rather unlike the discrete increments in current map servers.) The user is able to open any number of windows, and the windows are linked to each other in a tree hierarchy by restrictions on zooming—a parent must be more zoomed out than any of its children—and by the black rectangles in the parent that show the locations of its children. Each window is independently zoomed and panned; buttons and a slider are provided for zooming, and panning is accomplished by dragging the mouse.

As an example of the responsive behavior, when the user drags the mouse to pan a window, the system immediately translates the image that is displayed in that window. This is done without consulting either the server or the local cache, so it should happen within the 0.1 second level of interaction. Corresponding to the 1 second and 10 second levels of interaction, the local cache is searched and relevant data is displayed, and new data may be downloaded from the server. (The analogy is suggestive, not literal—we certainly don't expect the system to take 10 seconds to query the server.) The multi-threaded architecture allows these activities to take place concurrently.

At the bottom left of the screen is the display "W L U P R". These letters light up in red to inform the user what the system is doing. The most important is the W, which when lit indicates that the client is waiting for data to arrive from the server. The others are for

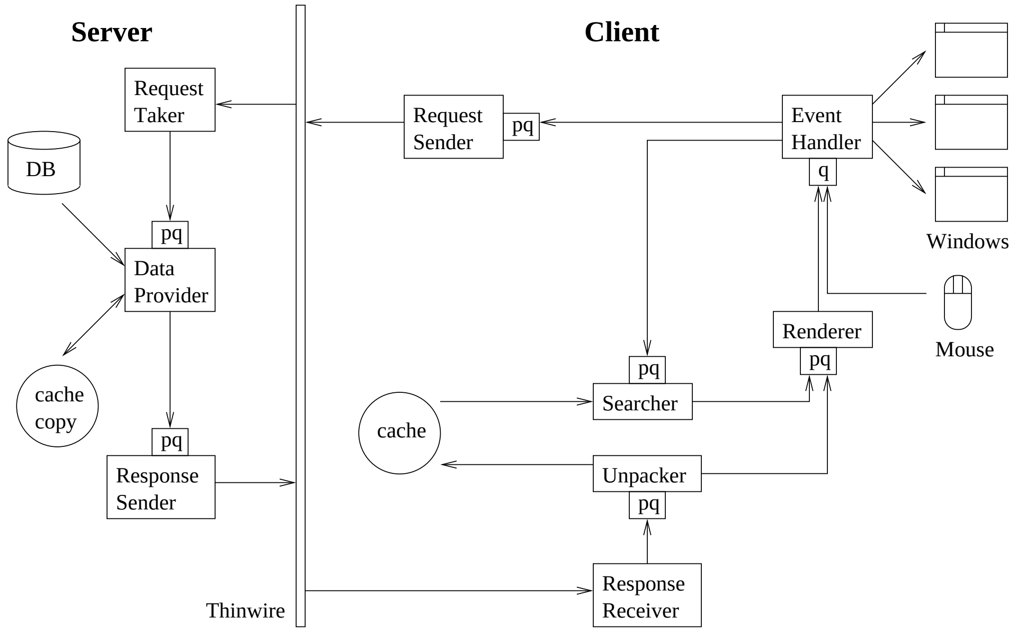

Figure 2: Runtime System Data Flow Diagram

loading, unpacking, processing and rendering the data. It is important to note that while W is lit the system does not freeze, but responds normally to the user, using local data and even possibly getting responses from the server for newer requests. This functions as the indicator, as mentioned in the previous section, that assures the user that the system is still working on any requests that may be taking a long time.

The data we use is the U.S. Census Bureau's TIGER data [39], generalized into eight levels of detail. Within each level the data is grouped into small connected regions, and the data for each region is compressed and stored in the database as a single *blob*. The blob is the unit of data exchanged between client and server; a request comes to the server as a geographic range, and any blob that intersects that range is returned to the client (unless the client already has it).

A much more detailed description of the application and issues of data preprocessing is in [1], and an early version was described in [40]. As far as we know, this is the first web-based visualization system with smooth zooming and panning for such a large geographic dataset—a zoom factor, from highest overview to highest detail, of almost 10,000. A demo is available on the web at `http://cs.nyu.edu/visual/demos/tigerDemo`.

3 Thread Architecture

Figure 2 illustrates our multi-threaded client-server architecture, with the flow of data and requests through the system. The large rectangles represent threads, and the small rectangles labeled "pq" or "q" represent priority queues or queues, the data structures that handle inter-thread communication and task prioritization. We use the term "task" to refer to either a request or data which is encapsulated as a message and passed from thread to thread, and also to refer to the procedure that a thread executes upon receiving such a message. The diagram is partitioned into a client program (on the right) and a server program (on the left). The basic flow of data and requests begins with user input from the mouse on the right to the Event Handler thread. Requests flow generally leftward to the server, and responses back to the client and eventually output to the windows. Much of the complexity and sophistication of this architecture, for providing responsiveness, is encapsulated in the priority queues, which are described in detail in section 4. (Where there is no ambiguity, we sometimes abbreviate "priority queue" with just "queue".) For the duration of a client connection, each thread repeatedly executes a loop in which it gets an input task from its queue or priority queue, or the thinwire, and executes various operations as part of that task. A detailed description of the thread behaviors is in [1].

The sequence of events that follows from the user making a movement of the mouse, either for a pan or a zoom event, includes three data paths that together represent a complete response to the request:

1. *mouse → Event Handler → windows*
   This path represents the initial, immediate response to any mouse event; it does not involve querying the server or the local cache. The Event Handler acknowledges input with some minimal update to the screen, such as translating or scaling the on-screen image.

2. *mouse → Event Handler → Searcher → Renderer → Event Handler → windows*
   This is the data path for finding and displaying any data that has already been downloaded from the server. The Event Handler passes the request to the Searcher, which pulls the relevant data out of the cache and passes it to the Renderer. The Renderer prepares the data for rendering, and passes each object individually (one at a time) to the Event Handler, which finally displays them on the screen.

3. *mouse → Event Handler → Request Sender → Request Taker → Data Provider → Response Sender → Response Receiver → Unpacker → Renderer → Event Handler → windows*
   In this third and slowest path, the request is sent over to the server, where the Data Provider searches the database and sends the relevant data back to the client, but storing the information in the server's cache copy first. Back at the client, the Unpacker stores the data in the local cache, before sending it on to the Renderer for eventual display.

Note the parallel between the three data paths and the three levels of interaction discussed in section 1. The first data path corresponds to the 0.1 sec level of interaction, the second to the 1 sec level, and the third to the 10 sec level.

Each thread spends most of its active time (i.e., time when it is not 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.

For any given request, the first path completes before the second and third are started. The second and third paths for one request execute concurrently, and may also be concurrent with the first path for another request. There is also pipelined concurrency among the requests on each of the second and third paths.

Consider the four threads that read or write the thinwire. If there were no thinwire—if the client and server were on the same machine—then these threads would be unnecessary: the Event Handler would directly access the priority queue for the Data Provider, and the Data Provider would directly access the priority queue for the Unpacker. The thinwire necessitates these threads for two reasons. (1) Since messages must cross the thinwire in first-in first-out

(FIFO) order, the client cannot directly write the Data Provider's priority queue, but must involve an intermediary thread. Similarly for the server writing the Unpacker's priority queue. The Request Taker and Response Sender function as these intermediary threads. (2) Since the thinwire may be arbitrarily slow, we expect that any thread that writes the network will spend a lot of time in a blocked state. This would be especially bad for the Event Handler, since we need it to be available to handle mouse input. Having the Request Sender and Response Sender allows the Event Handler to continue handling mouse events and the Data Provider to continue searching the database, even while the client or server is blocked on writing the network, thus increasing parallelism.

The queue that the Event Handler uses for its input is part of the Java runtime environment, and we cannot replace it or modify it. Since the Event Handler gets both display tasks and mouse input tasks from the same queue, we must be careful about how we use that queue. If a large number of render tasks are put onto the queue, then any new mouse event that occurs will have to wait until all those tasks are completed 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, and so we try to offload as much work as possible from the Event Handler onto the other threads.

4 The Priority Queues

The priority queues provide two primary functions: inter-thread communication and task scheduling. 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 data requests, are associated with a particular window, and are partitioned into priority classes, one per window. If class i has higher current priority than class j then all tasks in i have higher priority than all tasks in j.

2. 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 classes maintain their previous ordering.

3. Within a priority class, tasks are prioritized according to the timestamp (or sequence number) of the request that the task is associated with. Later timestamps have higher priority.

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

Section 4.1 describes the behavior of the basic priority queue, and section 4.2 describes special capabilities required by the Response Sender priority queue.

4.1 The Basic Priority Queue

The basic priority queue includes the following data structures:

- `globalQueue`: an ordinary queue, for the global tasks.
- One `heap` for each priority class, storing tasks heap ordered by timestamp.
- `priorityClassList`: a move-to-front list, where each node stores a priority class id.

The priority queue also contains two hash tables, to provide constant time access to the heaps and into `priorityClassList`.

Following is a list of the priority queue supported operations. Each of these is executed as an atomic operation, so it is important that they be as fast as possible. Let n be the number of priority classes, and let m be the size of the largest heap at any given time.

- `getNextTask()`: return the highest priority task, and remove it from the priority queue. If `globalQueue` is non-empty then return its front task, else check the heaps in priority class order and return the top task on the first non-empty heap. Requires time $O(n + \log m)$.
- `addTask(id, timestamp, task)`: add the given task to the appropriate heap, with the given timestamp. Requires time $O(\log m)$.
- `addGlobalTask(task)`: add a task to `globalQueue`. Requires constant time.
- `addPriorityClass(id)`: add a new priority class to the front of `priorityClassList`, giving it the highest priority. Requires constant time.
- `removePriorityClass(id)`: remove a priority class from `priorityClassList`, and discard all tasks associated with it. Requires constant time.
- `flushToGlobalAndRemove(id)`: move all tasks from the given priority class, in heap order, to `globalQueue`, and then remove the priority class. Requires time $O(m \log m)$.
- `moveToFront(id)`: move the given priority class to the front of `priorityClassList`. Requires constant time.
- `clear(id)`: cancel all tasks for the given priority class by clearing the heap. Requires constant time.
- `invalidate(id)`: mark each task on the given heap as "invalid". Requires time $O(m)$.

Window specific tasks are passed from one thread to the next with `addTask()` and `getNextTask()`. Usually there are no global tasks and `getNextTask()` will return the highest priority task for the window that currently has input focus.

When the user opens a window, closes a window, or gives input focus to a window, `addPriorityClass()`, `removePriorityClass()`, or `moveToFront()`, respectively, is executed on all of the priority queues in the system. The flow of each of these commands through the system is not identical, though. For `addPriorityClass()` and `moveToFront()`, the Event Handler executes the operation on each of the client side priority queues, then sends a message to the server (via the Request Sender), at which point the Request Taker executes the command on each of the server side priority queues. The `removePriorityClass()` command must take a different path, for two reasons: (1) for cache consistency (which is discussed further in section 7), any data that has been stored in the cache copy must eventually make it into the cache, which means that tasks should not be removed from the Response Sender and Unpacker priority queues; (2)

requests on the Data Provider queue must be invalidated rather than cleared or discarded (see below). Therefore the Event Handler executes the `removePriorityClass()` operation on only three of the client side priority queues—all but the Unpacker's—before sending the message on to the server. The Request Taker executes `invalidate()` and `flushToGlobalAndRemove()` on the Data Provider's priority queue, and puts a *remove* message onto its `globalQueue`. From that point, the message works its way back to the Unpacker, but at each priority queue the `flushToGlobalAndRemove()` operation is performed. This way, the priority class is removed only after all unfinished tasks for its associated window have been processed.

When all unfinished tasks for a window have become irrelevant because of the latest user actions, those tasks are dropped with either `clear()` or `invalidate()`. Since we cannot remove tasks from the Response Sender, Unpacker or Data Provider priority queues, the Event Handler executes `clear()` only on the other three client side priority queues and the Request Taker executes `invalidate()` on the Data Provider priority queue. The trigger for this is application dependent; in our application it happens when the user zooms in or out or jumps to a new location in the geography.

The reason we use `invalidate` instead of `clear` is because we want the client to give an indication to the user whenever requests have been sent to the server but not responded to yet. This is how we meet the visualization requirement that the system show a "working on it" indicator whenever a response takes more than a second. For this to work the client must get a response to every request, even if the response is empty, which means that tasks cannot be cleared from the Data Provider queue. Invalidating them informs the server that it should send back an empty response.

4.2 The Response Sender Priority Queue

Suppose the Data Provider gets a request R_1 that intersects blob B, which is sent to the Response Sender. While B is on the Response Sender queue, the Data Provider gets a new 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, but B should really be *promoted* to the highest priority of any request that intersects it. The extended functionality of the Response Sender priority queue allows this to happen.

If R_1 and R_2 are in the same priority class—i.e., were generated by the same window—we want to associate B with R_2; if not, the relative priorities of R_1 and R_2 may change whenever the user gives input focus to a different window, so we should associate B with *both* requests, but make sure that it only gets sent to the client once.

To accomplish this, we introduce a new data structure, `idNodeMap`, a hash table that maps each blob id to a list of heap nodes that contain (a reference to) that blob. (There will be only one node per heap for any blob.) Now the `addTask()` operation must add an entry to `idNodeMap` for every blob task added, a constant time addition. For `getNextTask()`, when the task removed is a blob task, we must invalidate that blob in all heaps other than the one we are removing it from. 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 $O(n + \log m)$. Finally, we must define a new operation, `upgrade(blobId, priorityClassId)`, which the Data Provider executes for every blob B that has previously been sent, and that intersects a new request R. The `upgrade(B, C)` operation is this: (1) If B is not in `idNodeMap`, do nothing (it has already been sent to the client); else (2) search the list from `idNodeMap`, and if B is in `heap[C]`, then upgrade the node's priority in that heap to the priority of R, and "bubble"the node up to its proper place in the heap; else (3) add B to `heap[C]` and add the new heap node to the list for B 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 $O(n + \log m)$.

It is possible that the upgrade functionality would also benefit the Response Receiver queue. We have not implemented this 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.

5 Responsiveness—Formal Arguments

The complexity of our responsiveness requirements makes it difficult if not impossible to prove the optimality of our approach, or even define "optimality" in this context. Nonetheless, we can make the following two claims whose justifications shed some light on how the system works, and why it is responsive. The first claim looks at the response time for the minimal initial acknowledgment for any user input. A "render task" refers to a task send from the Renderer to the Event Handler for rendering to the screen.

Claim 1 *The initial acknowledgment of a mouse input cannot be delayed by any more than (1) a single render task; and (2) the initial acknowledgment of any previous mouse inputs that have yet to be acknowledged.*

Justification: The algorithm for the Renderer ensures that once it has put a task onto the Event Handler queue it will not put another task onto that queue until the first task is finished. So no more than a single 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. □

We want the initial acknowledgment to happen within 0.1 second. Practically speaking, it should therefore be sufficient to ensure that the sum of the time for a single render task plus the time for one (or maybe two) mouse input handling routines be under 0.1 second.

In the next claim we look at how much a task is delayed by other tasks. But first some definitions. For the purposes of this claim, we consider two types of delay to be *necessary*: (1) the time for a thread to finish a task it has started; and (2) the time to add a task to a priority queue. We define "normal operation" of the system to include sending requests and receiving responses, but not removing a window or other event that causes the the normal prioritization rules to be set aside. And finally, define *ideal execution environment* to be one in which each thread executes on a dedicated processor and there is no thread management overhead.

Claim 2 *Under normal operation in an ideal execution environment, the blobs associated with a request will arrive to the client without being delayed any more than is necessary.*

Justification: For blobs that are already at the client when the request is made, 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 for adding the task to the priority queue and waiting for the Request Sender to finish a previously started task, both of which are necessary. 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, if the blob has not already been sent, it 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, 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 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 theoretically 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 in section 4.2, 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. Also, if we assume that events that lead to setting aside the normal prioritization rules are relatively rare, they should not have a significant impact on perceived responsiveness.

Also notice that `addTask` is logarithmic time and performed in memory, whereas each thread along the path from the client to the server and back is primarily occupied with I/O. Therefore, it is likely that the thread that reads a queue will be occupied with I/O while a task is added to its priority queue, so that the two types of necessary delay frequently occur in parallel.

6 Responsiveness—Testing

Because of the unique nature of our system design, with all the task reordering and cancellation, and the unique nature of our goal—responsiveness—it is not obvious what a testing procedure should be measuring. 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, we are not getting an idea of how the system performs while in flux, which is essential to responsiveness. If we measure the response time for individual requests, we are not measuring responsiveness—sometimes we prefer a slower response to certain request, because other more urgent requests come along that take higher priority. What we really want is to have the *right* data being transferred at the *right* time.

We have devised three simple measurement methods that give a rough indication of three different aspects of the responsiveness. The experiments were run on a DSL connection with a measured download bandwidth of 510 Kbps and a rated upload capacity of 128 Kbps. The server computer is a 500 MHz Sun Netra T1 AC200 with 1 GB of memory running Solaris 2.8. The database management software is PostgreSQL. The client computer is a 1.6 GHz Pentium 4 PC with 256 MB of memory running Linux.

The first test gives a sense of the quality of the dynamic task prioritization. The method is to measure the response time for each individual request, and then partition the responses into three categories: (1) responses that are for the most recent request (i.e., the user has made

	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	5.1 sec	5.9 sec	14%
DC, Bos., NY	6.2 sec	8.4 sec	26%
SF, DC, Bos., NY	8.1 sec	10.7 sec	24%
Phil., SF, DC, Bos., NY	9.6 sec	14.0 sec	31%

Table 1: Times between clicking the first jump button and system coming to rest.

no more requests since making the request that led to this response); (2) responses that, while not most recent, are still "relevant", which in our application means that the data should be currently shown in one of the windows; and (3) data that is no longer relevant, which in our application means that it is not visible in any window. If the dynamic task prioritization is working well, the first category of responses should be the fastest and the third category should be the slowest; this is indeed what we have found. We ran three different user sessions, and overall the average response time was 277 msec for the most recent requests, 363 msec for the "still relevant" requests, and 778 msec for the "no longer relevant" requests.

The second measurement method involves some very simple tests by hand that demonstrate the effects of task cancellation. The results are shown in table 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—recall the "W" light described in section 2.) 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 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.

The third measurement is related to the first claim in the previous section: we want the initial acknowledgment to a mouse input to be within 0.1 second, and this is related to the time for a single render task and the time the Event Handler spends on a single input event. This is highly application dependent—and even implementation and platform dependent. For our geographic visualization system, we have measured the time for a single render task, which is the rendering of a polygon along with some dependent polygons, and on our test platform most polygons can be rendered within 20 msec, and only the very largest polygons require close to 100 msec.

7 Cache Management

Cache management turns out to be a tricky problem because of the task reordering and cancellation. We have already seen in section 4 how cache considerations prevent us from cancelling tasks in certain situations; here we consider how to coordinate the client and server when the cache needs to be pruned. Since tasks can be reordered at any time, data is not added to the cache and cache copy in the same order, so it is not immediately obvious that the caches are always consistent, or even what is the meaning of "consistent" in this context. In section 7.1 we look at some flawed strategies, then we describe our approach and prove its correctness in section 7.2.

7.1 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 should be shown on the screen. When the system is at rest, we should have $C = C'$ and $S \subseteq C$. We assume that the client and server execute the same pruning algorithm, though which actual blobs are pruned depends on the state of the cache or cache copy at the time it is pruned. It seems natural to have the client decide when to initiate the pruning process, since it knows its own memory usage and what the user's current viewpoint is.

The naive approach is to have the client prune the cache, then send a *prune* message to the server, then the server prunes the cache copy. This introduces a consistency problem. Suppose we start with $b \in C$, $b \in C'$, and $b \notin S$. The Unpacker initiates a pruning, and removes b from the cache, so now $b \notin C$. The user then moves the current view so that $b \in S$. The Request Sender happens to send the corresponding request for data to the server before the Unpacker passes on the prune message, so the server does not return b because we still have $b \in C'$. When b is finally pruned from cache copy we have $b \notin C$, $b \notin C'$, and $b \in S$. At this point the system can come to rest with $S \not\subseteq C$, which means that some data will seem to have disappeared from the dataset.

Now suppose we start with $b \notin C$ and $b \notin C'$. First the client sends a request that covers b to the server. Next the client prunes the cache with an algorithm that would prune b if it were there, and sends a *prune* message to the server. The Data Provider receives the request first, and sends b in response after storing it in the cache copy, but then receives the *prune* message and removes b from the cache copy, so we still have $b \notin C'$. But when b arrives to the client we'll have $b \in C$ and $b \notin C'$, and the system can come to rest. 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 initiates the prune and chooses a pruning algorithm (e.g., delete all objects outside region R), but delays doing the actual prune until it notifies the server and the server prunes the cache copy. This protocol still runs into a problem because of task prioritization. Suppose *prune* has a high priority, and we start with $b \notin C$ and $b \notin C'$. In response to a request, the Data Provider adds b to the cache copy, and puts it onto the Response Sender priority queue. Now $b \in C'$. Next the Data Provider receives a *prune* command, prunes b from the cache copy, and puts the *prune* command onto the Response Sender priority queue. Now $b \notin C'$ again. The Response Sender processes the prune command first, because it has higher priority, sending it to the client. The client executes the prune algorithm, which would remove b if it were in the cache. Finally the Response Sender sends b to the client where it is added to the cache, and the system can now come to rest with $b \in C$ and $b \notin C'$. Similarly, if the *prune* command has low priority, we can have the system come to rest with $b \notin C$ and $b \in C'$.

7.2 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. For this, we introduce a new operation on the priority queues: `flushToGlobalAndAdd(task)`. This operation moves all tasks for all priority classes to `globalQueue`, in order of priority, and then adds the given task to `globalQueue`. It requires time $O(nm \log m)$. In our case the task we will add at the end is the prune command. 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.

So here, finally, is the working protocol:

1. The Unpacker chooses a prune algorithm A, and sends a prune(A) command to the server.
2. The Data Provider executes prune(A) on the cache copy.
3. The Data Provider calls `flushToGlobalAndAdd(prune(A))` on the Response Sender queue.
4. The Response Receiver calls `flushToGlobalAndAdd(prune(A))` on the Unpacker queue when receiving prune(A).
5. The Unpacker executes prune(A) on the cache, but only after putting all blobs that came before prune(A) into the cache.

Before proving the correctness of this protocol, we must define what it means to be "correct". Informally, what we really want is that if the server should send a blob to the client, then it does, and if it shouldn't, it doesn't. What we show below is that (1) for every request that arrives at the Data Provider, all blobs that intersect the request will at some time in the future be in the cache together, and (2) the Unpacker never receives a blob that duplicates one already in the cache.

First, we make some definitions. Let prune_i be the i-th prune command *initiated* by the Unpacker—and only later *executed* by the Data Provider and then Unpacker. 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 $\text{add}_i(x)$, for some x. Before getting to the theorems, we establish some intermediate results.

Lemma 1 *(1) Every* op_i *that occurs at the Data Provider also occurs (later) at the Unpacker, assuming the number of user requests is finite; and (2) Every* op_i *that occurs at the Unpacker also occurred (earlier) at the Data Provider.*

Proof: (1) 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. (2) The Unpacker only executes operations that come from the Response Receiver, the Response Receiver only sends commands that it receives from the Response Sender, the Response Sender only sends commands that it receives from the Data Provider, and the Data Provider only sends op_i commands after executing them.

□

Lemma 2 prune_i *precedes* op_j *at the Data Provider iff* prune_i *precedes* op_j *at the Unpacker.*

Proof: Lemma 1 guarantees that the set of operations at the two locations are the same, so the lemma statement is well defined. When the Data Provider processes prune_i, it executes `flushToGlobalAndAdd(prune)` on the Response Sender queue, so that 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 thinwire is FIFO, so the Response Receiver will receive op_j before prune_i iff op_j preceded prune_i at the Response Sender. When the Response Receiver processes prune_i, it executes `flushToGlobalAndAdd(prune)` on the Unpacker queue, so that the Unpacker will receive op_j before prune_i iff op_j preceded prune_i at the Response Receiver. □

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

Proof: By definition, prune_i is initiated before prune_j. At every priority queue, the prune command is placed onto its `globalQueue`, which is a FIFO structure, and the thinwire is a FIFO structure. Therefore, at the Data Provider the prune commands arrive in the same order in which they were initiated, so prune_i precedes prune_j. From this and lemma 2, the relationship also holds at the Unpacker. □

Let C_i and C_i' be the set of blobs added to the cache and cache copy, respectively, before prune_i, and after prune_{i-1} if $i > 1$. Let P_i and P_i' be the set of blobs removed from the cache and cache copy, respectively, as part of prune_i. Let S_i and S_i' be the set of blobs in the cache and cache copy, respectively, immediately after prune_i, or the empty set if $i = 0$. Then we have the following lemma.

Lemma 4 *For* $i \geq 1$, $C_i = C_i'$, $P_i = P_i'$, *and* $S_i = S_i'$.

Proof: Lemma 3 implies that the sets C_i and C_i' are well defined. $C_i = C_i'$ then follows from lemma 2. Since for each i the same prune algorithm is performed on the cache and cache copy, $P_i = P_i'$ follows from $C_i = C_i'$ by induction on i. $S_i = S_i'$ follows from $C_i = C_i'$ and $P_i = P_i'$ by induction on i. □

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 be the set of blobs in the database that intersect a request R. If the user makes only a finite number of requests, then after the Data Provider receives R there will be a time when all $b \in B$ are in the cache, and that time will be before the next prune, if there is one.*

Proof: Consider any $b \in B$. At the time that R arrives to the Data Provider, b is either in the cache copy, or not. Case 1: If b is not in the cache copy, $\mathrm{add}(b)$ will be executed on the cache copy, and by lemma 1 $\mathrm{add}(b)$ will later be executed by the Unpacker. By lemma 2, that will happen before the next prune, if there is one. Case 2: If b is in the cache copy, 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: If there has been a prune, let prune$_i$ be the last prune before the Data Provider received R. Since clearly $b \in S_i'$, $b \in S_i$ by lemma 4, and will remain in the cache at least until the next prune after R, if there is one. Case 2.2: If there has not been a prune, then, by lemma 2, add(b) will be performed by the Unpacker, before the next prune, if there is one. In every case, then, b is 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 1, the Data Provider previously executed the add(b) command, and by lemma 2 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, $b \notin S_i'$, and by lemma 4, $b \notin S_i$. Since there can be only one add(b) command between any two prunes, b cannot have been added after prune$_i$ but before this add(b), so b was not in the cache when the Unpacker received the add(b) command. □

8 Related Work

Forman's Petra-Flow [11] is a general programming framework for building responsive applications, though not specifically for 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. Tasks may be cancelled when they become obsolete. The sample applications built with the framework were a database front-end, a Mandlebrot fractal explorer, and a photo album Web browser. Each of these applications uses a relatively small number of tasks with straightforward priority relationships.

Under HTTP, the Hypertext Transfer Protocol, a client requests web resources and the server returns those resources. In HTTP/1.0 [3] each request/response transaction requires its own TCP connection. To reduce connection overhead, many HTTP implementations allow persistent connections, in which multiple transactions take place in sequence over a single connection. In HTTP/1.1 [10] "pipelining" is also allowed, wherein 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. Nielsen, et al. [30] found that HTTP/1.0 with up to six concurrent connections gave faster responses than a single HTTP/1.1 connection without pipelining, but HTTP/1.1 with pipelining had the best response times.

Schmalstieg and Gervautz inserted the network into the traditional rendering pipeline to make the Remote Rendering Pipeline [35, 34]. The client keeps a cache of downloaded data so that the server and network portions of the pipeline can sometimes be short-circuited, but in other respects it follows the rendering pipeline approach: data for each frame goes from the database, through stages such as visibility culling and rendering, to the final image.

Cohen-Or and Zadicario [9] reduced the number of frames that need to be sent across the network by having the server send an ϵ-superset of the geometry in a frame. In the approach of Cohen-Or [8], the client can extrapolate the next few frames from a reference frame, and the server only needs to send a difference image, preventing the accumulation of errors. (See also [29].) Both of these approaches take advantage of the nature of the walkthrough—continuous changes to the user view, and standard 3D geometry. Teler and Lischinski [38] also make use of these properties to drop frame rate strictures altogether, at least in the server and network portions of the system. Using occasional updates on user motion from the client, the server predicts where the user is going, and (ideally) sends the data ahead of time. Zach and Karner improve on their algorithm by optimizing for the worst case scenario, when the prediction is wrong, perhaps because the user suddenly changes direction.

The ViRGIS (Virtual Reality GIS) project [32, 33, 31] focuses on data storage and retrieval techniques appropriate for geographic data, while the client-server communication is relatively straightforward queries on a remote database. Two other projects, that of Gerlhof and Kemper [20] and Knafla [25, 26], designed multi-threaded architectures to facilitate prefetching pages from remote databases, but not specifically for visualization. In both cases, the main application thread on the client side, which handles the user interface, will block if the page it needs has not been downloaded yet, which means that system responsiveness is dependent on the quality of the prefetching technique.

Chang, Yap and Yen [7] and Chang [5] designed a thinwire system for visualization of very large "multi-foveated" images using wavelet techniques. The client requests various pieces, at various resolutions, of the underlying image. The client has a multi-threaded architecture that allows it to give some kind of response to every request, no matter how slow the network, but client-server communication is synchronous, so that user requests must be handled one at a time by the server, with no cancelling or reordering. Yap and Yen [41] later improved the client side architecture so that pipelining is possible, and similar to HTTP, the server must respond to each request in order. The improved client also allows some request cancellation, but the server must handle every request fully, and there is no reordering. The wavelet and foveation techniques introduce intra-data dependencies, especially between different levels of detail, which limits the amount of task cancellation that can be done.

9 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, where bandwidth between user and data is highly volatile and unpredictable. Even if bandwidth temporarily drops to zero, the system continues to respond to user requests intelligently and promptly, using whatever information it has available.

Instead of the traditional approach of designing to maintain a given frame, we have adopted a paradigm for responsive design based on the human conversation. This approach allows us to build a system that has fewer restrictions and resource requirements, but still feels responsive. The system architecture has three data paths built in for three levels of response, which suggests the three levels of interaction that humans perceive: the 0.1 second level, for perception of immediate cause and effect relationships; the 1 second level, the time for an unprepared response, and the pace of human dialogue; and the 10 second level, the time for a unit of cognitive work. The system also allows user actions to lead to task cancellation and re-prioritization, which suggests the kind of interruptions that often take place in conversation.

There are a number of directions this research can take in the future. Most importantly, we would like to apply these responsiveness techniques to different datasets and environments. One useful extension would be to add a layer of sparse, dynamic data, such as traffic flow or movement of truck or rail shipments. Applying these techniques to data with dependencies, such as the multi-foveated image data, is another interesting problem. We would like to incorporate responsiveness into more general distributed systems, rather than just client-server, and incorporate some of the benefits of distributed systems, such as fault tolerance, into our responsive system.

As was discussed in section 6, 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 of Harel, et al. [21, 22, 23], 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. These include using prefetching (see [20, 25, 26], for example), more sophisticated task scheduling (see [6] for a discussion of online scheduling based on a similar thinwire visualization problem for large images), and multiple socket connections between client and server.

References

[1] Kenneth Been. *Responsive Thinwire Visualization of Large Geographic Datasets.* PhD thesis, New York University, 2002. `http://cs.nyu.edu/visual/home/pub/`.

[2] Michael Bender, Ralf Klein, Andreas Disch, and Achim Ebert. A functional framework for web-based information visualization systems. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):8–23, January-March 2000.

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

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

[5] Ee-Chien Chang. *Foveation Techniques and Scheduling Issues in Thinwire Visualization.* PhD thesis, New York University, 1998. `http://cs.nyu.edu/visual/home/pub/`.

[6] 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.

[7] 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.

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

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

[10] 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`.

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

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

[13] 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.

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

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

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

[17] 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.

[18] 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.

[19] 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.

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

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

[22] 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.

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

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

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

[30] 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.

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

[32] 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.

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

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

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

[36] 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.

[37] 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.

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

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

[40] 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.

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