A Progressive Vector Map Browser

José Augusto S. Ramos[1], **Claudio Esperança**[2], **Esteban Walter G. Clua** [1]

[1] Instituto de Computação – Universidade Federal Fluminense (UFF)
Niterói, RJ – Brazil

[2]Programa de Engenharia de Sistemas – COPPE – Universidade Federal do Rio de Janeiro (UFRJ)
Rio de Janeiro, RJ – Brazil

ja_sapienza@yahoo.com.br, esperanc@cos.ufrj.br, esteban@ic.uff.br

Abstract. *With the increasing popularity of web-based map browsers, remotely obtaining a high quality depiction of cartographic information has become commonplace. Most web mapping systems, however, rely on high-capacity servers transmitting pre-rendered tiled maps in raster format. That approach is capable of producing good quality renderings on the client side while using limited bandwidth and exploiting the browser's image cache. These goals are harder to achieve for maps in vector format. In this work, we present an alternative client-server architecture capable of progressively transmitting vector maps in levels-of-detail (LOD) by using techniques such as polygonal line simplification, spatial data structures and, most importantly, a customized memory management algorithm. A multiplatform implementation of this system is described, as well as several performance and image quality measurements taken with it.*

1. Introduction and related work

The increasing popularity of web-based systems and services for delivering maps can be regarded as one of the most important developments in the advancement of cartography. Several aspects of these systems have merited investigation in recent years, such as the improving reliability of the Internet and web server infrastructure, ascertaining the quality and fidelity of the served data, coping with privacy and security issues, maximizing the use of screen space, and making rational use of the available bandwidth [Burghardt et al. 2005].

One important design decision when building a web mapping application is the choice between raster and vector formats. Some of the most popular systems, such as Google Maps [Google, Inc. 2008] or Yahoo Maps [Yahoo! Inc. 2008], are mostly raster-based, which means that they serve pre-rendered digital images, that is, maps are first rendered in several resolutions in the server, cut into blocks (called tiles), and then sent to clients based on the desired area and zoom level. The key reasons for using rasters are: (1) all work spent in authoring a good quality representation can be done on the server, with the merely composing a big picture from several small image tiles; (2) the transmission of raster data of fixed size uses limited bandwidth, and (3) web browsers already manage image caches and, thus, little or no memory management is needed on the client side.

Several web mapping systems have also been proposed which use vector data as well. Perhaps the most widely used system is the MapServer open source project [University of Minnesota 2008]. The technology for serving maps in vectorial form

has been intensely researched (see [Kraak and Brown 2001] for a survey). The advent of the SVG (Scalable Vector Graphics) has further propelled these initiatives and general information and software for deploying such systems is abundant (see [Carto:net - cartographers on the net 2008], for instance). An adequate solution for the problem clearly depends on the use of techniques for hierachically organizing vector data according to its visual importance, obtaining what is usually known as level-of-detail data structures [Davis 2008]. At the same time, some sort of spatial indexing is usually required for quickly retrieving data which overlap the region of interest [Gaede and Günther 1998]. Several authors have also investigated suitable techniques for memory caching of spatial data with and without prefetching [Stroe et al. 2000, Doshi et al. 2003], as well as methods appropriate for handling multi-resolution vector data [Chim et al. 1998].

One less studied aspect of web cartography is the relation between the level-of-detail of the data being served, the use of bandwidth and client memory management, specially for vector-based software. In particular, most systems assume that all clients have the same (small) amount of memory at their disposal and, as a consequence, statically link the level-of-detail of the served map to the size of the area being viewed.

In this paper we describe data structures and algorithms which make it possible to remotely deliver and present high-quality vector maps in a progressive manner, making efficient use of the available bandwidth, and adapted to the memory profile of any given client without encumbering the communication protocol with information about client memory state. In particular, although the server receives from the client only information pertaining to the area being viewed, it is able to guess and progressively transmit only needed data.

2. Overall system architecture

According to [McMaster and Shea 1992], around 80% of the total data in vector geographical databases are polygonal lines. This statement guides the scope of the proposed architecture: (1) only vector maps with open or closed polygonal lines are considered, and (2) the use of network bandwidth is optimized by restricting the transmission of line data with just enough detail for a faithful representation.

Thus, we propose a client-server system for serving map data containing a program (the server) capable of directly accessing all polygonal lines of a given map – from a database, for instance – and progressively sending it to interactive visualization applications (the clients). Clients have limited memory capacity and thus store only enough line data so as to present a good depiction of the map within a visualization window. Each time a user changes this window, the contents of the client memory must be updated by requesting relevant data from the server and discarding unneeded information.

The system preprocesses all polygonal lines comprising the map into two hierarchical data structures, which can be quickly traversed in order to obtain the needed information. The two structures used are: (1) a spatial index, which is needed to prune out polygonal lines which do not contribute to the current viewing window, and (2) a structure for organizing the vertices of each polygonal line in order of visual importance – the so-called level-of-detail (LOD) data structure. It should also be mentioned that the present architecture does not handle polygonal line importance classification, i.e., it is

considered that all polygonal lines intersecting a given window need to be drawn at some level of detail. Although map visualization applications typically provide some way of establishing which lines can be left out when rendering the map at certain zoom levels, we do not concern ourselves with this feature in this paper.

Both server and client process the viewing window change in a similar manner. The client only needs to inform the server of the new viewing window in order to receive the needed data not yet stored in its memory. This requires that the server is kept aware of the memory state of each client: if there are n lines in a map, the server maintains for each client an array of n integers which map each line to the level of detail in which it is represented in the client's memory. Whenever new vertices need to be sent from server to client, this transmission is broken into blocks of limited size. In other words, the architecture supports progressive transmission of detail information so that the visual quality of the client images improve over time, at a rate that depends solely on the available bandwidth.

3. Server-side preprocessing

For each map being served, their polygonal lines must be submitted to a preprocessing step in order to (1) build a hierarchical level-of-detail data structure for their vertices, and (2) build a spatial index used to support window queries.

3.1. Level-of-detail data structure

We employ the well-known Douglas-Peucker (DP) line smoothing algorithm [Douglas and Peucker 1973]. Although it is not a particularly fast algorithm, running in $O(n \log n)$ at best and $O(n^2)$ in the worst case [Hershberger and Snoeyink 1992], it is ranked as the best when it comes to preserving the shape of the original line [McMaster 1987]. Furthermore, it produces a hierarchical representation which can be used for level-of-detail processing.

The DP algorithm recursively subdivides a polygonal line by selecting the vertex at greatest distance from the line segment defined by the first and last point. Figure 1 illustrates this process for an example line. Observe that the subdivision process can be

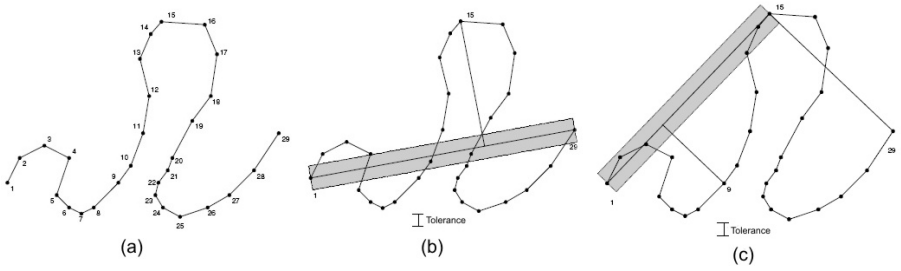

Figure 1. Douglas-Peucker line simplification (adapted from [Casanova et al. 2005] (a) A polygonal line with 29 vertices; (b) vertex 15 is furthest from line segment 1–29, (c) polygonal lines 1–15 and 15–29 are processed recursively.

registered in a binary tree, where each node N corresponds to a subdivision vertex and the corresponding distance d_N, whereas its left and right sons correspond to sub-trees for

the left and right sub-chains. Thus, an approximation within tolerance ϵ for the polygonal line can be extracted by visiting the tree in pre-order and pruning out branches rooted at nodes with distances $d_N < \epsilon$.

In this work, we are interested in obtaining increasingly finer representations for each polygonal line. This can easily be implemented by quantizing tolerance values into an integer range $[1, MaxLOD]$. A coarse representation will thus be assigned to level-of-detail (LOD) 1 by visiting the tree in pre-order for tolerance ϵ_1. Vertices in this representation are then marked with $LOD = 1$. The process is repeated for increasingly smaller tolerances ϵ_i for $i = 2 \cdots MaxLOD$, and in each stage i, non-marked vertices are labeled with the corresponding $LOD = i$ value. An important consideration in this process is that, ideally, the number of vertices marked for each LOD value should be approximately constant, so that transmitting the next finer representation of a given line (see constant δ in Section 4) can be done in constant time. In our implementation, ϵ_1 is chosen as the distance in world coordinates corresponding to the width of a pixel for a fully zoomed out projection of the map, while successively finer tolerances were estimated by setting $\epsilon_{i+1} = 0.8\epsilon_i$.

3.2. Spatial indexing

In theory, the worst case scenario for vector map browsing consists of setting the viewing window so that all polygonal lines are enclosed in it. In practice, however, users frequently are interested in investigating a small portion of the whole map. It stands to reason, therefore, that some spatial indexing method be used for selecting polygonal lines intersecting any given query window.

Although the present work does not focus on the issue of efficient spatial indexing, we surveyed several works in the field (see [Samet 2006] for a comprehensive compilation) and chose the relatively simple Expanded MX-CIF Quadtree [Abel and Smith 1984] data structure for speeding up window queries. This is a data structure for rectangles which, in the context of this work, correspond to each polygonal line minimum enclosing bounding box. Each rectangle is represented in the data structure by a collection of enclosing quadtree blocks. In our implementation, this collection contains a maximum of four blocks, although other amounts might also be possible. The four blocks are obtained by determining the minimum enclosing quadtree block, say B, for each rectangle, say R, and then splitting B once to obtain quadtree blocks B_i ($i \in \{\texttt{NW, NE, SW, SE}\}$) such that R_i is the portion of R, if any, that is contained in B_i. Next, for each B_i we find the minimum enclosing quadtree block, say D_i, that contains R_i. Now, each rectangle is represented by the set of blocks consisting of D_i (refer to Figure 2 for an example). Window queries can be easily computed by means of a recursive descent algorithm on the tree, i.e., start with the root and recursively visit sons if their quadrants intersect the given window.

4. Memory Management

In the context of a client-server system, the issue of memory management should be governed by the following considerations.

Memory capacity: It is assumed that the client memory is bounded by some given constant. At any one time, the client has its memory partially occupied with a subset of

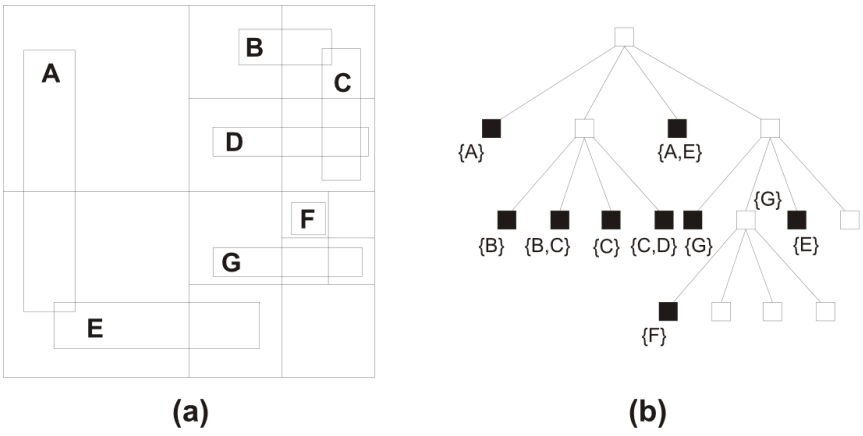

Figure 2. Example MX-CIF Quadtree (b) and the block decomposition induced by it for the rectangles in (a) (adapted from [Samet 2006]).

the map's polygonal lines at some level-of-detail. When the user changes the viewing window, i.e., performs a zooming or panning operation, the memory contents should be altered if it does not contain a "good" representation of the map as revealed by the newly defined window.

Memory control protocol: When requesting new data from the server, some agreement must be reached on what data is needed. In other words, the server must either be told, or must already know what data to transmit to the client in response to a user action. Thus, there are two general approaches for the control protocol: (1) the client requests the needed data items, meaning that the server does not know the client memory's contents, or (2) the server is aware of the memory management operations performed by the client by simulating the same operations as stipulated by a fixed algorithm. Clearly, the former approach uses up more bandwidth than the latter. On the other hand, CPU usage could be greatly increased if the server is to reproduce operations of all clients. In this work, we adopt the second strategy, where the increase in time complexity is alleviated by employing a relatively simple memory management rationale which can be executed in tandem by both server and client.

In order to describe our approach, let us first define a few terms. Let

- M be the maximum client memory size;
- m be the amount of data that can be transmitted from the server to the client in one transaction, i.e., in time for the client displaying the next frame;
- $S = \{L_i\}$ be the current resident set, i.e., the set of polygonal lines L_i that intercept the current viewing window W;
- W be the current viewing window;
- $LOD(L)$ be an integer in $[0, MAXLOD(L)]$ representing the level-of-detail of polygonal line L. It is assumed that if $L \notin S$, then $LOD(L) = 0$;
- $BestLOD(L, W)$ be an estimate for the "best" level-of-detail for exhibiting a polygonal line L in viewing window W. Clearly, this function should return 0 if L is not visible within W. Otherwise, it should return a LOD which is adequate

for the current screen resolution. For instance, it should be just fine enough for mapping two consecutive vertices to different pixels; and

- δ be an estimate of how much memory associated with increasing or decreasing a level-of-detail step for any given polygonal line L. In other words, on average, a polygonal line L should occupy approximately $\delta \times LOD(L)$ memory.

Then, adjusting the items resident in memory requires two classes of operations: operations that increase and operations that decrease the use of memory. Operations in the first class cause data to be transferred from the server to the client. We distinguish two of these:

1. *IncreaseLOD*(L) increases the level-of-detail for polygonal line L. This means that if $LOD(L) = k > 0$, then after its execution $LOD(L) = k + 1$.
2. *Load*(L) brings polygonal line L from the server in its coarsest form. As a precondition, $LOD(L) = 0$ and after its execution, $LOD(L) = 1$.

The second class corresponds to operations which cause data to be thrown away from client memory. Observe that these operations do not cause any network traffic between server and client. We also define two operations of this class:

1. *DecreaseLOD*(L) decreases the level-of-detail for polygonal line L.
2. *Unload*(L) unloads polygonal line L from memory altogether.

Thus, any memory management algorithm will consist of sequentially performing these operations in some order in a timely manner and without infringing memory limits M and m. Our algorithm uses two heaps I and D which hold operations of each of the two classes described above. A crucial consideration is how to define the ordering between operations in each heap. Clearly, operations of type *Load* should have a higher priority than all operations of type *IncreaseLOD*. Similarly, operations of type *DecreaseLOD* should have higher priority than operations of type *Unload*. In our implementation, the ordering between operations *IncreaseLOD* for two lines L_1 and L_2 depend on how distant the $LOD's$ of each line are from their estimated "best". In other words, we use $|BestLOD(L) - LOD(L)|$ as a priority measure. The priority between operations *DecreaseLOD* is defined in a similar way. Algorithm 1 DefineOperations describes how the two heaps are created.

Once the operation heaps are known, client and server process them in parallel. Operations are executed subject to the memory and bandwidth restrictions discussed above. Algorithm 2 ExecuteOperations summarizes the rationale for operation execution. It is important to realize that executing an operation has different meanings for client and server. For instance, executing an *IncreaseLOD* operation in the client entails receiving line detail from the server and updating the geometry for that line, while for the server it means merely sending the additional vertices. Similarly, while *DecreaseLOD* entails updating the polygonal line data structure for the client, the server needs only to take note that the corresponding memory was freed in the client. A limitation of Algorithm 2 is related to the fact that δ is merely an estimate of the amount of memory associated with decreasing or increasing the LOD of any given line. This may lead to $|S|$, the amount of memory used for the polygonal data, eventually exceeding M if the newly received LOD data is bigger that δ. This, in general, is not a problem since the overflow should be small on average. In any case, the restriction can easily be lifted by assigning to δ a sufficiently large value.

Algorithm 1: DefineOperations

Input: W_{new}: the new window set by the user
Output: I and D: heaps containing operations which cause memory increase/decrease

```
begin
    I ← ∅
    D ← ∅
    S' ← set of lines which intersect W_new
    for L ∈ S' ∪ S do
        if L ∉ S then
            Enqueue [Load, L] in I
        if L ∉ S' then
            Enqueue [Unload, L] in D
        if LOD(L) < BestLOD(L, W_new) then
            for i ← LOD(L) + 1 to BestLOD(L, W_new) do
                Enqueue [IncreaseLOD, L] in I
        else if LOD(L) > BestLOD(L, W_new) then
            for i ← BestLOD(L, W_new) + 1 to LOD(L) do
                Enqueue [DecreaseLOD, L] in D
end
```

Note that the scheme described above is easily adapted to support progressive transmission. Suppose that Algorithm 2 terminates with a non-empty heap I. Then, if the viewing window for the next frame is unchanged, there is no need to run Algorithm 1 again, and the next execution of Algorithm 2 will further process heaps I and D, thus providing an increasingly finer rendering of the map, as illustrated in Figure 7.

5. Implementation and Results

A prototype implementation of the framework described in this paper was built and used to conduct several experiments in order to assess the validity of our proposal.

The development was supported by the following tools: user interfaces were built with version 4.4.1 of the multi-platform Qt [TrollTech 2008] library and the *Shapelib* library v. 1.2 was used for reading *Shapefiles* [MapTools 2008]. The pre-processor was written in C++ and compiled using version 4.1 of the gcc compiler [Free Software Foundation Inc. 2008]. The client and server programs which implement the algorithms described above were written in Python (version 2.5.3) with the Qt library accessed by means of the the PyQt wrapper [Riverbank 2008] version 4.4.3. Communication between clients and server use the XML-RPC specification, a protocol for Remote Procedural Call (RPC) coded in XML [UserLand Software, Inc. 2008]. It is important to remark that all of these tools are Open Source and, thus, freely available.

The pre-processing described in Section 4, was carried out with a dedicated program which (1) reads polygonal map data in *Shapefile* format, (2) executes of the Douglas-Peucker algorithm and computes the level-of-detail hierarchy for each polygonal, (3) cre-

Algorithm 2: ExecuteOperations

```
Input: I and D: heaps containing memory management operations
begin
    t ← 0
    while I ≠ ∅ and t < m do
        if |S| + δ > M then
            [op, L] ← Dequeue from D
            execute op(L)
        else
            [op, L] ← Dequeue from I
            execute op(L)
            t ← t + δ
end
```

Table 1. Sequence of map browsing operations used in the experiments.

Frame	0	8	12	14	16	19	21	23	25	27	31	36	39	41	43	45
Op.	ZF	Z+	P	Z+	Z+	Z+	P	P	P	Z+	Z+	Z-	Z-	Z-	Z-	ZF

ates an extended MX CIF Quadtree for supporting window queries, and (4) saves the important information into a structured XML (eXtensible Markup Language) file which is used as input for the server program.

The system deployment is straightforward, requiring only that a server process is started in some computer and one or more client processes in the same or some other machine connected by a TCP/IP network. When initialized, the server will load the XML generated in the preprocessing stage. When a client connects to the server, it informs its cache memory size and transmission block size, i.e., constants M and m discussed in Section 4. The server then sends a reply message containing the coarsest possible map representation and a compressed representation of the MX-CIF data structure. After this initialization stage, the communication protocol proceeds as described in the above sections.

Experiments were conducted using a test map with the state limits of Brazil in scale 1:1.000.000 produced by the Brazilian Geography and Statistics Institute (IBGE) [IBGE 2008]. This map contains 349 polygonal lines, each with 173 vertices on average. The preprocessing stage produced an Extended MX CIF Quadtree of height 6 and containing 172 leaf nodes, while polygonal lines were split into up to 25 LOD steps.

The maximum client memory size M was set to 32 KBytes, while the maximum block size per frame was set to 2 KBytes. A client was then used for browsing the test map by issuing a fixed sequence of zooming and panning operations. These are shown in Table 5, where ZF, Z+, Z- and P stand for zoom full, zoom in, zoom out and pan, respectively.

The first experiment aimed at measuring the use of client cache memory during the browsing session. The chart in Figure 3 shows that, as expected, the use of mem-

ory increases almost linearly and, around frame 19, levels out at the maximum memory capacity.

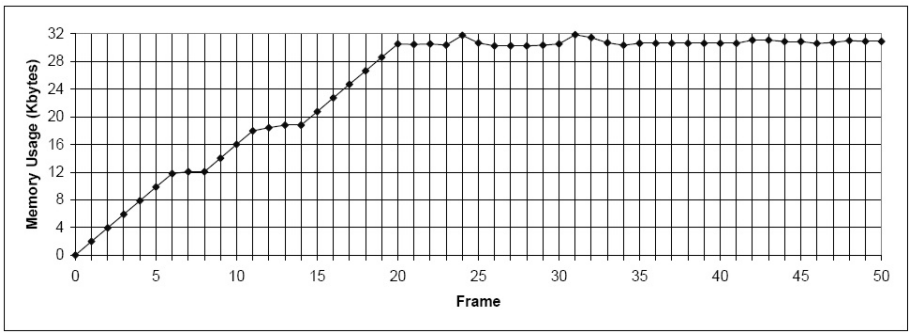

Figure 3. Client cache memory usage per frame.

The second experiment gauged the usage of bandwidth by measuring the sizes of transmitted blocks. These numbers are shown in the chart of Figure 4. As expected, network usage is kept under the imposed limit of 2 KB per frame. Observe that successive frames with high bandwith usage – but without intervening browsing operations – correspond to progressive transmission and rendering of map data. We also note that zooming in and zooming to the full map generate more network traffic than panning or zooming out.

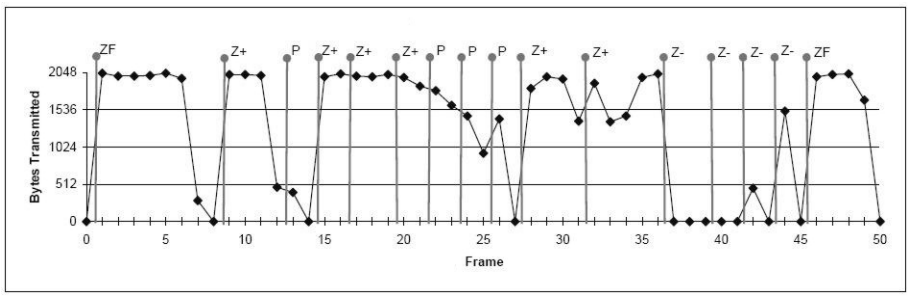

Figure 4. Bytes transmitted per frame.

It is also useful to observe the frequency of cache memory management operations as a function of the frame number. The chart in Figure 5 plots two lines, one depicting the cache inclusion operations, i.e., those that generate network traffic, and another depicting the cache exclusion operations. Note that no exclude operations are required before frame 19, as there is still enough room for storing polygonal data. After this point we observe that the number of include and exclude operations increase and decrease in sync. This is reasonable since exclude operations are required to make room for include operations. Another important observation is that the number of operations does not necessarily follow the pattern of bandwidth usage. This can be attributed to the fact that the amount of data for each LOD step is not constant.

Finally, it was sought some way for measuring the picture quality observed in the client as a function of time (frame). For this purpose, we considered that a given polygonal line L present in window W is rendered perfectly if it is represented in cache

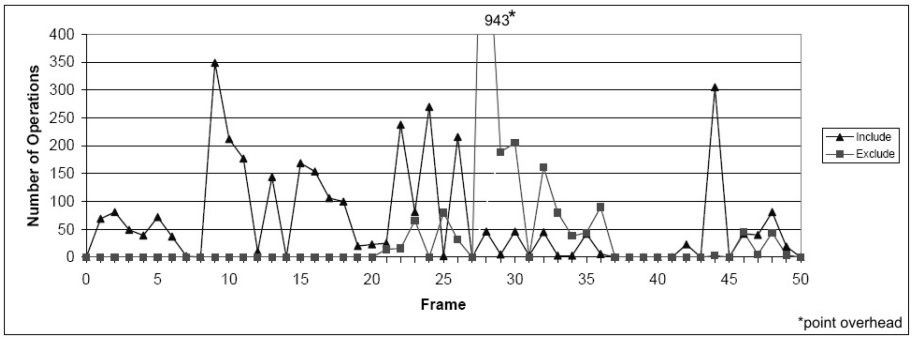

Figure 5. Cache memory management operations per frame.

memory with LOD $BestLOD(L, W)$ or greater. Thus, a percentage measure of image quality Q may be estimated by

$$Q = \frac{100}{|R|} \times \sum_{L \in R} \frac{\min(LOD(L), BestLOD(L, W))}{BestLOD(L, W)},$$

where R is the set of lines intersecting W. A plot of this quality measure is shown in Figure 6. It is possible to observe that after a few frames the system achieves maximum quality and never falls significantly below 80%. Obviously, this threshold is dependent on the relationship between the total map size and M, the cache memory size. Similarly, the latency observed for reaching maximum quality depends on the allowed block size m.

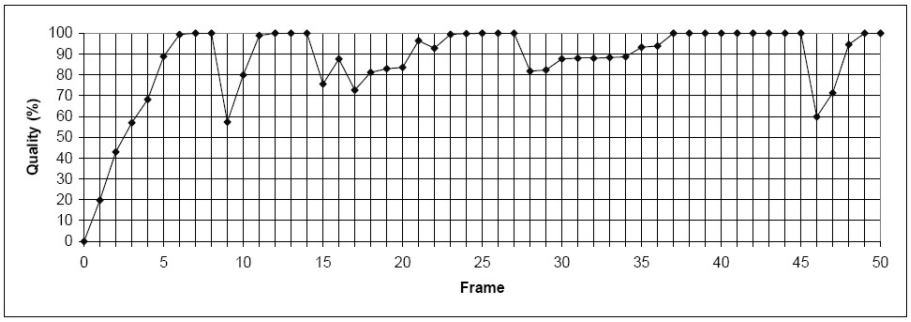

Figure 6. Image quality per frame.

As an example of quality increasing over time is shown in Figure 7. The user performs a zoom operation in a window displaying a map with all states of Brazil, causing the system to perform a progressive transmission and rendering until it reaches 100% of image quality.

6. Conclusions and suggestions for future work

The client-server framework for remotely displaying vector maps described in this work was designed to achieve several goals: be simple, scalable, make predictable use of network bandwidth and support progressive transmission and rendering. The prototype implementation and admittedly limited experimental evidence seem to indicate that these objectives were largely met. A continuation of this work would necessarily start by a more thorough experimentation, with the use of different data sets and other values for

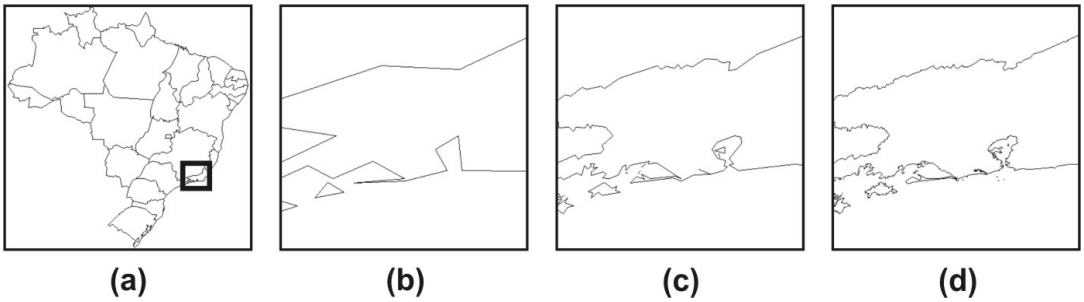

Figure 7. A zoom-in operation performed on the black rectangle (a) causes progressive transmission and rendering (b), (c) until the system achieves maximum image quality (d).

constants M and m. Ideally, a performance evaluation should also be attempted in order to evaluate the scalability of the server.

Clearly, a production system would require the addition of several improvements such as visual importance classification and a more fine-grained processing of polygonal data so that a given polyline could be stored with varying levels of detail in client memory. A complete system would probably also include the ability to serve raster data.

References

Abel, D. J. and Smith, J. L. (1984). A data structure and query algorithm for a database of areal entities. *Australian Computer Journal*, 16(4):147–154.

Burghardt, D., Neun, M., and Weibel, R. (2005). Generalization services on the web – a classification and an initial prototype implementation. *Proceedings of the American Congress on Surveying and Mapping – Auto-Carto*.

Carto:net - cartographers on the net (2008). Svg, scalable vector graphics: tutorials, examples, widgets and libraries. `http://www.carto.net`.

Casanova, M., Câmara, G., Davis, C., Vinhas, L., and Queiroz, G. (2005). *Bancos de dados geográficos*. Editora Mundo GEO, Curitiba.

Chim, J. H. P., Green, M., Lau, R. W. H., Leong, H. V., and Si, A. (1998). On caching and prefetching of virtual objects in distributed virtual environments. In *MULTIMEDIA '98: Proceedings of the sixth ACM international conference on Multimedia*, pages 171–180, New York, NY, USA. ACM.

Davis, C. (2008). Geometria computacional para sistemas de informação geográfica. `http://www.dpi.inpe.br/~gilberto/livro/geocomp/`.

Doshi, P. R., Rundensteiner, E. A., and Ward, M. O. (2003). Prefetching for visual data exploration. In *DASFAA '03: Proceedings of the Eighth International Conference on Database Systems for Advanced Applications*, page 195, Washington, DC, USA. IEEE Computer Society.

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

Free Software Foundation Inc. (2008). GCC, the GNU compiler collection. http://gcc.gnu.org/.

Gaede, V. and Günther, O. (1998). Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231.

Google, Inc. (2008). Google maps. http://maps.google.com.

Hershberger, J. and Snoeyink, J. (1992). Speeding up the Douglas-Peucker line-simplification algorithm. In *Proc. 5th Intl. Symp. on Spatial Data Handling*, pages 134–143.

IBGE (2008). IBGE - Instituto Brasileiro de Geografia e Estatística. http://www.ibge.gov.br.

Kraak, M.-J. and Brown, A. (2001). *Web Cartography – Developments and prospects*. Taylor & Francis, New York.

MapTools (2008). Shapefile C library v1.2. http://shapelib.maptools.org/.

McMaster, R. B. (1987). Automated line generalization. *Cartographica*, 24(2):74–111.

McMaster, R. B. and Shea, K. S. (1992). *Generalization in digital cartography*. Association of American Geographers, Washington, D.C.

Riverbank (2008). Pyqt4 download. http://www.riverbankcomputing.co.uk/software/pyqt/download.

Samet, H. (2006). *Foundations of Multidimensional and Metric Data Structures*. Morgan-Kaufman, San Francisco.

Stroe, I. D., Rundensteiner, E. A., and Ward, M. O. (2000). Scalable visual hierarchy exploration. In *DEXA '00: Proceedings of the 11th International Conference on Database and Expert Systems Applications*, pages 784–793, London, UK. Springer-Verlag.

TrollTech (2008). Qt cross-platform application framework. http://trolltech.com/products/qt/.

University of Minnesota (2008). UMN MapServer. http://mapserver.gis.umn.edu.

UserLand Software, Inc. (2008). Xml-rpc homepage. http://www.xmlrpc.com.

Yahoo! Inc. (2008). Yahoo! maps. http://maps.yahoo.com.