Dynamic Simplification and Visualization of Large Maps

Nabil Mustafa*
Department of Computer Science
Duke University
nabil@cs.duke.edu

Shankar Krishnan
AT&T Labs – Research
krishnas@research.att.com

Gokul Varadhan*
Department of Computer Science,
University of North Carolina, Chapel Hill
varadhan@cs.unc.edu

Suresh Venkatasubramanian
AT&T Labs – Research
suresh@research.att.com

Abstract

In this paper, we present an algorithm that performs simplification of large geographical maps through a novel use of graphics hardware. Given a map as a collection of non-intersecting chains and a tolerance parameter for each chain, we produce a simplified map that resembles the original map, satisfying the condition that the distance between each point on the simplified chain and the original chain is within the given tolerance parameter, and that no two chains intersect.

In conjunction with this, we also present an out-of-core system for interactive visualization of these maps. We represent the maps hierarchically and employ different pruning strategies to accelerate the rendering. Our algorithm uses a parallel approach to do rendering as well as fetching data from the disk in a synchronous manner. We have applied our algorithm to a gigabyte sized map dataset. The memory overhead of our algorithm (the amount of main memory it requires) is output sensitive and is typically tens of megabytes, much smaller than the actual data size.

1 Introduction

Cartographic Map Generalization is "the process of deriving from a detailed source spatial database a map or database the contents and complexity of which are reduced, while retaining the major semantic and structural characteristics of the source data appropriate to a required purpose" [37]. The process of generalization can be decomposed into various subtasks called *operators*. Examples of operators include *selection/elimination*, *simplification*, *smoothing* etc. Each operator contributes to visual clarity and efficiency at various levels of detail. We concern ourselves with the simplification operator in this paper.

Line simplification is a very important generalization operator, since a majority of the map features are represented as lines, or polygons composed of lines. Simplification reduces the amount of line detail, and thus contributes to a faster visualization. Furthermore, the maps created by modern imaging techniques are seldom optimized for rendering efficiency, and can usually be replaced with far fewer primitives (vertices and chains) without any loss in visual fidelity. In order to further improve the rendering speed and quality, it is desirable to compute several versions of these maps. More detailed versions can be used when the object is very close to the viewer, and are replaced with coarser representations as the object recedes. Often, it is desirable to adapt the level of refinement in different areas depending on the viewing parameters, yielding a 'dynamic level-of-detail' simplification strategy. Finally, the simplification algorithm can be *static* or *dynamic* depending on whether it is performed as a preprocessing step or as part of the rendering loop.

In this paper, we focus on the problem of *map simplification*. Given a map represented as a planar subdivision (arrangement of polylines), we wish to compute a simplified representation at varying levels of detail (depending on the current view point) in such a way that the shape features (the essential geometric and topological characteristics)

*This work was done while visiting AT&T Labs, Florham Park.

of the region are preserved. In order to achieve real-time performance, we implement key steps of our simplification algorithm using modern graphics hardware. Apart from significant performance gains, the use of hardware also allows us to handle the issues of data scalability and robustness much better than traditional geometric techniques.

In many instances, the maps that one might wish to render are very large; typical map data from the TIGER database [5] can exceed 1 GB, and the maps can have hundreds of millions of vertices. Such large maps cannot be stored completely in main memory at all times, and thus it is hard to render such maps interactively. We present an *out-of-core* map simplification and rendering system that, depending on the current viewpoint, only loads into main memory an appropriate portion of the map at an appropriate level of detail, thus allowing much higher frame rates than would be possible with unsimplified maps.

1.1 A Hybrid Vector/Raster-based Approach

Traditional methods of map generalization can be classified as either *vector* or *raster-based* methods. These refer to the way map data is represented: either as a collection of polygonal chains (or vectors), or in terms of a subdivided domain where each atomic element (a *pixel*) has quantitative information related to the map. Most (if not all) map simplification methods, going as far back as the Douglas-Peucker [9] line simplification method, employ the vector representation of data, and all of the work in computational geometry (see Section 3.2) does the same.

In our work, although our basic map representation is in the form of vector data, we employ transformations in and out of a raster-based representation (specifically, a representation in terms of pixels of a graphics buffer). The purpose of this is to take advantage of the tremendous improvements in speed achievable by modern-day graphics hardware. Especially in the context of interactive map simplification and visualization, such an approach, combined with the out-of-core methods we describe below, allows us to achieve real-time interactivity, which prior systems were unable to achieve.

The disadvantage of raster-based methods for map generalization is the inherent loss of geometry. An image-based representation of maps destroys topological information e.g. the exact representation of boundaries between regions. We minimize this by maintaining (simultaneously) both the vector and raster representations of a map. Using the underlying graphics engine, we compute key information needed to simplify the map, and then modify our vector data accordingly.

Novel features of our approach are:

Quality of simplification: Our algorithm guarantees that any two chains in the original map will not intersect in the simplified map. Unfortunately, our methods cannot prevent the boundary of a region from intersecting itself. We discuss ways of addressing this problem in Section 8.

Metric independence: In order to define the notion of shape preservation, any simplification algorithm must assume that the object to be simplified is embedded in a distance space. Our algorithm runs independently of the choice of distance function. This is important because notions of shape preservation can change depending on the application.

Constrained simplification: We can incorporate *point constraints* in addition to geometric and topological constraints. Consider, for example, we are given a map of the United States with cities marked as vertices inside the map. It is desirable that if we generate simplifications of this map, cities do not change states. This condition is equivalent to preserving *sidedness* of chains with respect to these vertices.

Out-of-core simplification: As discussed earlier, many maps are much too large to maintain wholly in main memory. We present an out-of-core simplification scheme that enables us to retrieve from disk only those parts of a map (appropriately simplified) that the user viewpoint dictates. Our prefetching strategy allows this to happen in parallel with the rendering phase, thus providing a seamless interactive experience.

Dynamic automated level-of-details: Our system allows the user to zoom in and out of regions of the map, with detail becoming more or less visible accordingly. Additionally, the system allows the user to view different regions of the map at different levels of detail. This provides the user with a great degree of control over the nature of the simplification; certain regions that are viewed as more important can be maintained at higher levels of detail than other, less important regions.

The paper is organized as follows. Section 2 formally defines our problem, related work is surveyed in Section 3. We briefly describe the overview of our approach in Section 4 and present a brief description of the graphics hardware in Section 4.1. The details of our simplification algorithm are given in Section 5. In Section 5.5 we present our method for dealing with point constraints. Section 6 presents our out-of-core simplification system. Performance results are presented in Section 7 and we present conclusions and future directions in Section 8.

2 Problem Definition

In this section, we define the problem of map simplification. The basic unit of a map is a *chain*. A chain $\mathcal{C} = \{v_1, v_2, \ldots, v_m\}$ of size size($\mathcal{C}$) $= m$ is a polygonal path, with adjacent vertices in the sequence joined by straight line segments (see Fig. 1). A *shortcut segment* for the chain $\mathcal{C}$ is a line segment between any two vertices v_i and v_j, $i \neq j$, of $\mathcal{C}$. Let $d(\cdot, \cdot)$ denote a metric on points in the plane. The results in this paper are independent of the metric, and therefore $d(\cdot, \cdot)$ could denote any metric. The distance $d(v_i, \overline{v_j v_k})$ between a point v_i and a segment $\overline{v_j v_k}$ is defined as the distance from v_i to the (infinite) line supporting $\overline{v_j v_k}$.

We define the *error* of the line segment $\overline{v_i v_j}$ with respect to $\mathcal{C}$ as

$$\Delta(\overline{v_i v_j}) = \max_{i \leq k \leq j} d(v_k, \overline{v_i v_j})$$

Let $\mathcal{C}' = \{v_1 = v_{j_1}, v_{j_2}, \ldots, v_{j_{m'}} = v_m\}$ be a chain whose vertices form a subsequence of the sequence of vertices of $\mathcal{C}$. Then we define the *error* of $\mathcal{C}'$ as

$$\Delta(\mathcal{C}') = \max_{1 \leq k < m'} \Delta(\overline{v_{j_k} v_{j_{k+1}}})$$

A chain is *self-intersecting* if any two of its line segments intersect. The *endpoint* vertices of the chain $\mathcal{C}$, vertices v_1 and v_k, are called the *start* and *end* vertices respectively. We assume in this paper that all chains are non self-intersecting. Two chains *intersect* if they share any vertex other than their endpoints, or if any pair of line segments in the chains intersect.

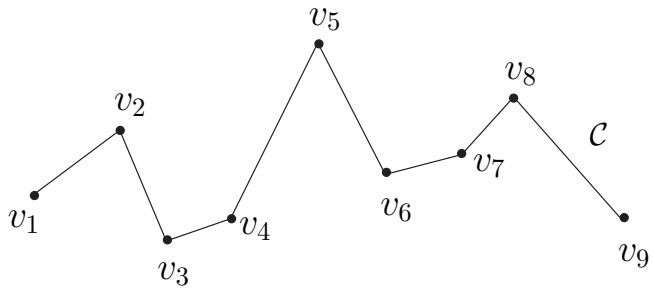


Figure 1: A single chain $\mathcal{C} = \{v_1, \ldots, v_9\}$ of size 9.

A map $\mathcal{M}$ is a set of *non-intersecting* chains. The *size* of a map is the sum of the sizes of its chains. The objective of map simplification is to take a map $\mathcal{M} = \{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n\}$ and a tolerance vector array $\{\epsilon_1, \epsilon_2, \ldots, \epsilon_n\}$ as input, and compute a map $\mathcal{M}' = \{\mathcal{C}'_1, \mathcal{C}'_2, \ldots, \mathcal{C}'_n\}$ such that

1. For each i, the vertices of $\mathcal{C}'_i$ are a subsequence of the vertices of $\mathcal{C}_i$ (and $\mathcal{C}'_i, \mathcal{C}_i$ share endpoints).
2. For each i, $d(\mathcal{C}'_i, \mathcal{C}_i) \leq \epsilon_i$.
3. For all $1 \leq i \neq j \leq n$, the chains $\mathcal{C}'_i$ and $\mathcal{C}'_j$ do not intersect.
4. The size of $\mathcal{M}'$ is minimum over all $\mathcal{M}'$ satisfying conditions 1,2 and 3.

If a chain $\mathcal{C}'_i = \{v_1 = v_{j_1}, v_{j_2}, \ldots, v_{j_{m'}} = v_m\}$ satisfies conditions (1) and (2) above, we say that it is a *valid simplification* of $\mathcal{C}_i$. See Fig. 2 for one valid simplification of the chain shown in Fig. 1. Note that condition (1) above implies that the line segments $(v_{j_1}, v_{j_2}), (v_{j_2}, v_{j_3}), \ldots, (v_{j_{m'-1}}, v_{j_{m'}})$ are shortcut segments of the chain $\mathcal{C}_i$. This fact will be crucial in our algorithm.

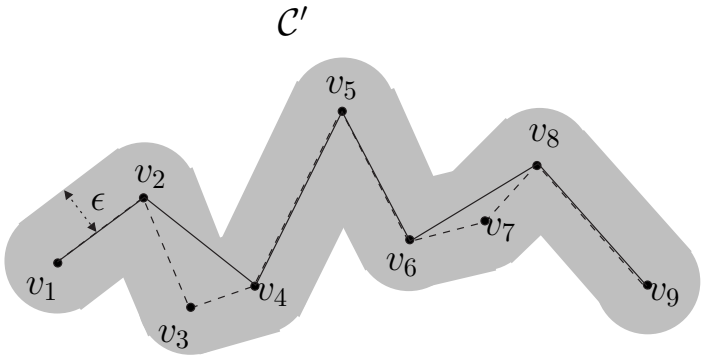


Figure 2: A simplified chain $\mathcal{C}' = \{v_1, v_2, v_4, v_5, v_6, v_8, v_9\}$ (solid line) of input chain $\mathcal{C}$ (dashed line), with $size(\mathcal{C}') = 7$, and $d(\mathcal{C}', \mathcal{C}) \leq \epsilon$.

Similarly, if all the chains in $\mathcal{M}'$ are valid simplifications of their corresponding chains in $\mathcal{M}$ and they satisfy condition (3) above, then we say that $\mathcal{M}'$ is a *valid simplification* of $\mathcal{M}$.

The distance function $d(\cdot, \cdot)$ expresses the similarity between maps. Intuitively, we wish to simplify the map as far as possible (expressed by the minimization constraint) while preserving the quality of the rendering (expressed by the distance constraint). It is important to note that there are two separate components of this similarity: geometry and topology. The topological constraint is expressed in the condition that the output set of chains is non-intersecting, and that each output chain has a corresponding input chain. The geometric constraint is expressed by the distance function $d(\cdot, \cdot)$. Most approaches for map simplification attempt to preserve one or the other; a simultaneous preservation of both constraints is hard.

3 Previous Work

We briefly survey previous work in the domain of map simplification. All the methods discussed here address the problem in its static formulation; to the best of our knowledge, there is no prior work that attempts to solve this problem interactively.

3.1 Cartography

There are many different line simplification methods: the reader is referred to the papers by McMaster [29, 30] for a detailed classification. In general, line simplification methods have differed in whether they are local in nature (examining local geometric properties of a chain to simplify it) or global.

One of the first and most effective methods for line simplification is the algorithm proposed by Douglas and Peucker [9] in 1973. This algorithm is very efficient, and it was observed by researchers [38, 28] that the simplifications produced by this method tended to conform to the subjective solutions produced by domain experts. A straightforward implementation of the algorithm runs in time $O(k^2)$ for a chain of size k; Hershberger and Snoeyink [19] show how to improve this to $O(k \log k)$.

Although the algorithm is simple and efficient, it can be applied only to single chains, and it is easy to see that when applied on an entire map (by applying it successively on individual chains), the resulting map is likely to have intersections. One solution to this problem, proposed by Estkowski [12], is to run the Douglas-Peucker algorithm first and then use a local improvement technique to eliminate intersections.

3.2 Computational Geometry

Map simplification (and line simplification) has been studied extensively by computational geometers [20, 8, 6, 17, 4]; Estkowski [11] has shown that map simplification is NP-hard. Moreover, Estkowski and Mitchell [12] have also shown that it is hard to obtain a solution to this problem that approximates the optimal answer to within a polynomial factor. For chain simplification, the best known method [6] obtains the optimal simplification in time $O(k^2)$, where k is the size of the original chain. However, none of the methods for chain simplification guarantee that the output chain will not intersect itself; ensuring that self-intersections do not happen (while preserving optimality) is an open problem and is suspected to be hard.

There are very few implementations of any of the above map simplification algorithms. To the best of our knowledge, there exists only one implementation (of the algorithm proposed by Estkowski [12]). This algorithm uses a local improvement heuristic to remove intersections. The timing results published in that paper indicate that on a typical map of size 100,000-200,000 it takes around 10-30 seconds on the machine we use for our experiments.

A variant of the above problem allows the introduction of new vertices that are not in the original chain (the so-called *Steiner* variant). This problem is also known to be NP-hard [17]; no non-trivial approximation algorithm is known for this variant.

3.3 Other Related Methods

There has been very little work in the graphics community regarding the problem of map simplification. However, the problem of polygonal mesh simplification [35, 33, 23, 22, 7, 13, 26, 32] has been studied extensively. We point the reader to the survey by Garland and Heckbert [13] for further details.

There have been some studies of exploiting the power of graphics hardware to implement geometric algorithms. Rossignac *et al.* [34] and Goldfeather *et al.* [14] use hardware to perform real-time constructive solid geometry. Greene *et al.* [15] have used the "Z query" feature of some specialized graphics hardware to implement their hierarchical Z-buffering algorithm. Krishnan *et al.* [24] use hardware to compute and visualize the depth contours of a point set, and Agarwal *et al.* [3] solve a number of geometric optimization problems in hardware. Hoff *et al.* [21] use the Z-buffering capabilities to compute Voronoi diagrams of dynamic primitives in real-time. We have used this technique in our system to perform our simplification.

A recent example of the use of hardware-assisted methods in the GIS community is the work by Guesgen *et al.* [16] to perform buffering operations on fuzzy maps for overlay operations.

4 Overview of Our Method

In this section, we briefly discuss the algorithmic concepts underlying the simplification algorithm. The basic approach is to start with some initial set of segments, throw away the ones that are "unsuitable" and build our simplification from the remaining segments.

As observed in Section 2, we need only consider shortcut segments from chains as possible line segments in a valid simplification. Thus, the problem is to determine which shortcut segments can appear in valid simplifications.

There are many ways to build an initial set of shortcut segments. For a chain of size m, there are $O(m^2)$ possible shortcut segments one could start with[1]. Once we have an initial set of candidate shortcut segments, we need to cull out invalid segments. For a shortcut segment to be valid, it must satisfy both geometric and topological conditions. All vertices on $\mathcal{C}$ that lie between the endpoints of a shortcut segment that might be present in a valid simplification of a chain $\mathcal{C}$ must be within distance ε of it. More precisely, a shortcut segment $\overline{v_j v_k}$ is said to be *proximate* if $\Delta(\overline{v_j v_k}) \leq \varepsilon$.

For shortcut segments s for chain $\mathcal{C}_i$ and s' for chain $\mathcal{C}_j$, we ensure that s and s' do not intersect. This will enable us to simplify chains independently of each other. Consider the Voronoi diagram of the set of chains in the

[1]For large maps this quadratic size can be prohibitive, and we can use a heuristic to choose a small set of "good" shortcut segments. We discuss this in Section 6.

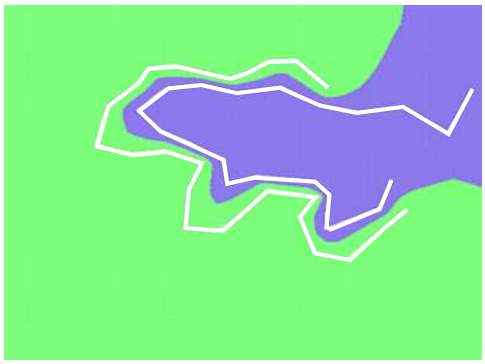


Figure 3: Voronoi regions of two chains that partition the plane.

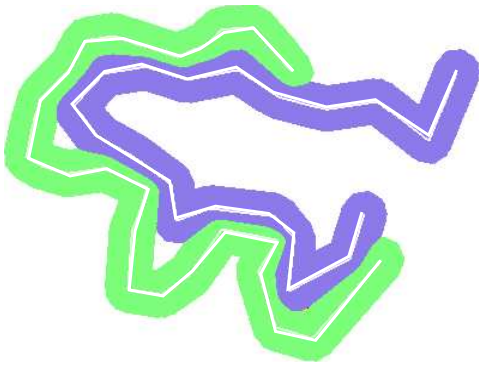


Figure 4: ε-Voronoi regions.

input map. Since Voronoi regions of non-intersecting chains partition the plane, two different shortcut segments lying inside their respective Voronoi regions cannot intersect. Thus, by requiring that shortcut segments must lie completely inside the Voronoi regions of their respective chains, we ensure that the desired condition is met. We call these regions the *compliant* regions. Shortcut segments lying inside their compliant regions are referred to as *compliant shortcut segments*. Fig. 3 shows the compliant regions of a map.

Now consider the region consisting of all points within distance ε of $\mathcal{C}$ i.e. the region $\mathcal{C} \oplus B_\varepsilon$, where $\oplus$, the standard Minkowski sum, is defined as $A \oplus B = \{a + b | a \in A, b \in B\}$, and B_ε is a ball of radius ε. Fig. 5 illustrates this region for a single chain. Any shortcut segment which does not lie completely inside this region is not proximate, though the converse does not hold, as illustrated by Fig. 5. We exploit this fact as following: instead of using the Voronoi diagram to define compliant shortcut segments, we use the ε-Voronoi diagram instead. The ε-Voronoi diagram is a Voronoi diagram with respect to the distance measure

$$d_\varepsilon(p, q) = \begin{cases} d(p, q) & d(p, q) \leq \varepsilon \\ \infty & \text{otherwise} \end{cases} \tag{4.1}$$

An example of ε-Voronoi diagram of two chains is shown in Fig. 4. The advantage of using ε-Voronoi regions is that they are subsets of the corresponding Voronoi region. Also, the nonproximate shortcut segments that do not lie within a region $\mathcal{C} \oplus B_\epsilon$ will also not lie in a ε-Voronoi region (due to the definition of the distance measure for ε-Voronoi region).

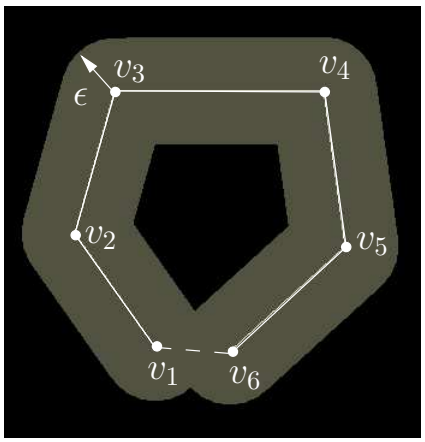


Figure 5: An example of a non-proximate segment $\overline{v_1 v_6}$ (dashed) completely inside the region $\mathcal{C} \oplus B_\varepsilon$.

4.1 The Graphics Rendering System

Modern graphics rendering systems are based on a pipeline architecture. In the first stage of this pipeline, the user gives the geometric primitives to be "drawn" to the evaluator. The basic primitive is a vertex, which consists of x, y, and z coordinates. Other primitives are lines, triangles, polygons and so forth. These primitives have various attributes such as texture, lighting etc. The second stage, per-vertex operations, performs various operations such as lighting, clipping, projection and viewport mapping on the input primitives. The third stage rasterizes the primitives, i.e. produces pixels from the geometric primitives which the graphics system will be able to draw. The fourth stage is per-pixel operations, such as blending and z-buffering. Finally, the pixels that pass the per-pixel operations of the previous stage are drawn, or *rendered*, in the frame buffers.

4.1.1 Frame Buffers

The frame buffer is a collection of several hardware buffers. Each buffer is a uniform two dimensional grid, composed of cells (one for each pixel). Let the number of pixels in each row and column be W and H respectively. Each pixel in a buffer is allocated a certain number of bits for data storage; thus the total memory in the frame buffer (the amount that is typically specified when advertising chips) is the product of WH, the number of buffers, and the number of bits per buffer. Three buffers that we make use of are:

Color Buffer. The color buffer contains the pixels that are seen on the display screen. It contain **RGB** color information and also contains information relating to the transparency of pixels, (also called the α value). Before a pixel is converted and rendered to a particular cell in the color buffer, it has to undergo various per-pixel operations. If it passes all the operations, it is rendered with the appropriate color. The color buffer typically allocates 8 bits per color per pixel [2].

Stencil Buffer. This buffer is used for per-pixel operations on a pixel before it is finally rendered in the color buffer. As the name suggests, it is used to restrict drawing to certain portions of the screen. We explain this in the next section. The stencil buffer typically allocates 8 bits/pixel.

Depth Buffer. The depth buffer stores the depth value for each pixel. Depth is usually measured in terms of distance to the eye, so pixels with larger depth-buffer value are (usually) overwritten by pixels with smaller values. More general operations can also be performed on the depth buffer, as described in the next section. The depth buffer typically allocates 24 bits/pixel.

The various buffers can be thought of as internal registers of the graphics engine. Operations are performed on the pixels in these buffers and the results are also stored in the buffers.

5 Details of the Algorithm

We now describe in detail our algorithm for map simplification. The input to the simplification algorithm is a map $\mathcal{M} = \{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$ consisting of n non-intersecting chains. A set of tolerance parameters, $\mathcal{E} = \{\varepsilon_1, \ldots, \varepsilon_n\}$ is also given, where the chain $\mathcal{C}_i$ has to be simplified with tolerance parameter ε_i. An example set of chains is shown in Fig. 6. We will use this example to illustrate the working of our algorithm at each step.

5.1 Step 1: Computing the ε-Voronoi Diagram

The first step in the algorithm is to compute the ε-Voronoi diagram of the set of chains, which is merely the Voronoi diagram of the chains with respect to the distance function d_ε defined in Equation (4.1). Note that for each chain, a different value of ε is used.

To do this, we make use of the hardware-based method for computing Voronoi diagrams as described by Hoff *et al.* [21]. Consider a set P of n points in $\Re^2$. Let cone(p) denote the $45°$-cone extending upwards in the z-direction from $p \in P$. This cone can be described as cone$(p) = \{(x, y, z) \mid (x - p_x) + (y - p_y)^2 = z^2, p = (p_x, p_y)\}$. A

[2] The latest graphics cards now allow upto 32 bits per color per pixel.

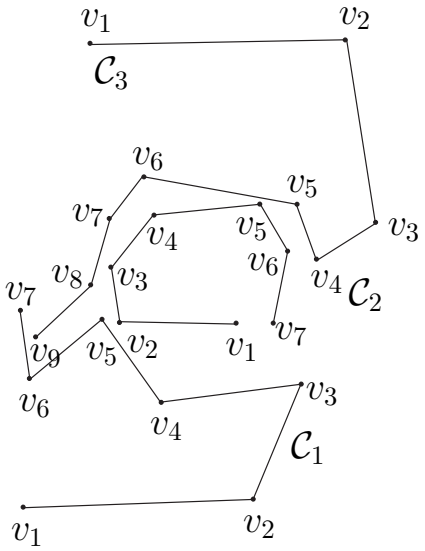


Figure 6: Input chains (non-intersecting) to demonstrate our algorithm.

standard result in computational geometry [10] states that the *lower envelope* of $\{\text{cone}(p)|p \in P\}$ is the Voronoi diagram of P.

Hoff *et al.* make use of this idea in their work. For each point $p \in P$, they construct an approximate cone comprising a fan of isoceles triangles. The number of triangles (and the apex angle of the triangle) are chosen in order to ensure that the error incurred (in approximating a cone by a set of triangles) is at most one pixel. Each such collection of triangles is assigned a unique color. When all the triangles are now rendered, they use the depth buffer to compute the depth at each pixel with respect to a viewpoint at $z = -\infty$. The depth buffer thus contains the lowest points in the arrangement of cones and hence the lower envelope. Since all cones are drawn in different colors, the color buffer contains a rendering of the Voronoi regions for the points. Furthermore, by using appropriate shapes, this idea can be extended to polygonal chains such as the ones in maps.

We modify their algorithm for computing the Voronoi diagram as follows: Instead of computing the Voronoi diagram in the color buffer, we will compute it in the *stencil buffer*, i.e. after the Voronoi diagram is computed, the stencil buffer gives the Voronoi diagram for the chains, where the region in the stencil buffer with value i is the Voronoi region for the chain $\mathcal{C}_i$. The stencil buffer can have values only up to 256. However, the technique works as long as the map is 256 colorable. In our system, we use simple heuristics to color the chains; it is conceivable that one might use an algorithm with formal guarantees [18] for most data sets. Further, by drawing the cones only to height ε, we obtain the ε-Voronoi diagram.

See Fig. 7 for an example of the stencil buffer state with the corresponding Voronoi diagram in the color buffer after this phase is finished, and Fig. 8 for the ε-Voronoi diagram of our three chains.

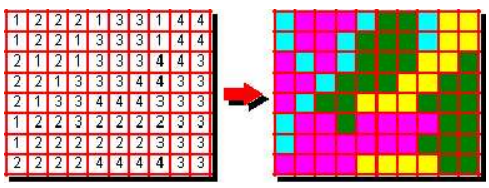


Figure 7: Stencil region with corresponding color buffer.

Algorithm 5.1 `Remove_NonCompliant_Segments`(Input Map $\mathcal{M}$)
Input: A Map $\mathcal{M} = \{\mathcal{C}_1, \ldots, \mathcal{C}_m\}$.
Output: The set of $compliant$ shortcut segments $\mathcal{S} = \{\mathcal{S}_1, \ldots, \mathcal{S}_m\}$.

```
S ← all shortcut segments in M
repeat
  for Chain C_i ∈ M do
    glStencilFunc( GL_FAIL, GL_FAIL, GL_PASS )
    glStencilTest( GL_NEQUAL, i )
    for Shortcut segment {v_j, v_k} ∈ S_i do
      Set(Color of {v_j, v_k})
      Draw({v_j, v_k})
    end for
  end for
  ColorBuffer = ReadColorBuffer( )
  for Unique color c in ColorBuffer do
    Remove segment whose color is c from S
  end for
until S does not change
```

5.2 Step 2: Computing Compliant Shortcut Segments

Once we have drawn the ε-Voronoi diagram on the stencil buffer, we can use it to find the set of compliant shortcut segments. Recall that a compliant shortcut segment lies completely inside the ε-Voronoi region of its chain.

For each chain $\mathcal{C}_i \in \mathcal{M}$, we first set the stencil test to pass only if the stencil value at a pixel is not equal to i. We then render all the shortcut segments $\mathcal{S}_i$ for the chain $\mathcal{C}_i$ with unique colors. This is done for all the chains. Note that if the number of shortcut segments, say $|\mathcal{S}|$, is greater than $2^{24} - 1$, the algorithm given in Algorithm 5.1 would have to be repeated $\lceil \frac{|\mathcal{S}|}{2^{24}-1} \rceil$ times, as we assume an 8-bit per channel color buffer (RGB).

Once we have drawn all the shortcut segments in $\mathcal{S}$ with unique color for all the chains, we read the state of the color buffer using the *glReadPixels()* call in OpenGL, and scan all the pixels. Based on the stencil test described above, it follows that only the shortcut segments $\mathcal{S}_i$ of chain $\mathcal{C}_i$ that go outside the ε-Voronoi region of $\mathcal{C}_i$ appear in the color buffer. Therefore, if the color of a shortcut segment is present in the color buffer, we immediately know that it is non-compliant because each shortcut segment is drawn with a unique color value. We remove the detected non-compliant segments from our original set $\mathcal{S}$. It is possible, however, that a non-compliant segment is occluded by another segment (or a group of segments). In this case, we may wrongly conclude that it is compliant. We solve this problem by repeating the process of segment elimination until the set $\mathcal{S}$ remains unchanged. For the maps tested, we typically needed two such repetitions.

The pseudocode for the algorithm is shown in Algorithm 5.1. Fig. 9 illustrates the algorithm by showing the initial set of shortcut segments (solid lines) for all the chains, the compliant shortcut segments (dashed lines) and the non-compliant ones (bold dotted lines).

5.3 Step 3: Removing remaining nonproximate shortcut segments

As observed in Section 4, there are cases (e.g. Fig. 5) where the previous phase cannot detect and remove all the nonproximate shortcut segments. For each shortcut segment $\overline{v_i v_j}$ we maintain its maximum distance ($\Delta(\overline{v_i v_j})$) to the original chain. For each compliant segment, we compare this value with the corresponding tolerance parameter for the chain and determine if it is proximate or not. Fig. 10 shows the proximate segments that are retained after this step.

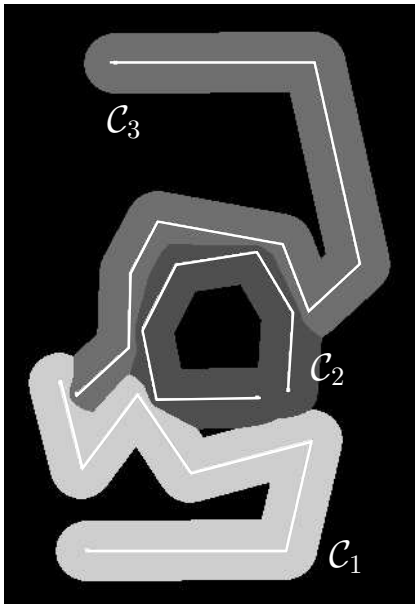


Figure 8: Step 1: ϵ-Voronoi diagram of the chain segments.

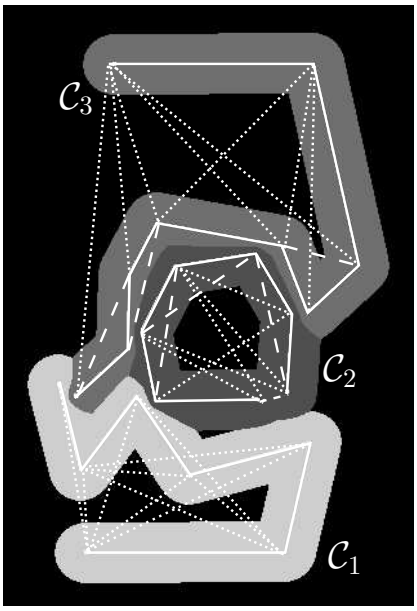


Figure 9: Step 2: Set of compliant shortcut segments (dashed) and non-compliant shortcut segments (dotted).

5.4 Step 4: Computing Shortest Paths

The output of step 3 gives us a set of compliant, proximate segments $\mathcal{S}_i$ for each chain $\mathcal{C}_i \in \mathcal{M}$. The next (and final) step is to find a shortest path from the start to the end of each chain using the segments in $\mathcal{S}_i$. Let $G_i = (V_i, E_i)$ be an undirected graph. For each vertex $v_j \in \mathcal{C}_i$, add the node n_j to V_i. Similarly, for each shortcut segment $\overline{v_j v_k} \in \mathcal{S}_i$, add the edge (n_j, n_k) to E_i.

Now we find the shortest path from the first node to the last node in G_i. Since G_i is unweighted, this amounts to doing a breadth-first search from the first node. We replace each node in the shortest path thus found with the corresponding vertex of the chain $\mathcal{C}_i$. Since each edge in the graph G_i corresponds to a compliant proximate shortcut segment in $\mathcal{S}_i$, the corresponding sequence of vertices we get forms a valid simplification $\mathcal{C}_i'$ of the input chain $\mathcal{C}_i$. Using a shortest path enables us to reduce the complexity of the simplification. Although this is optimal for the set of valid shortcut segments, it may occasionally lead to self-intersections.

Fig. 11 shows the final simplification produced by the algorithm.

5.5 Point Constraints

There are many situations where maps contain point features whose sidedness (with respect to boundaries) we wish to preserve. For example, a careless simplification of a map of Massachusetts could cause a feature corresponding to the city of Boston to end up in the Atlantic Ocean! Such conditions can be expressed by specifying that certain points must remain on the same side of certain boundaries as they were in the input. We call such constraints *point constraints*. A simplification where no point constraints are violated is said to *respect* the point constraints.

Our algorithm can be modified to compute simplifications that respect point constraints. Consider a point constraint $(p, \mathcal{C})$, where p is a point and $\mathcal{C}$ is the constrained chain. Then any simplification $\mathcal{C}'$ of $\mathcal{C}$ must have p on the same side as $\mathcal{C}$. The crucial fact is that a shortcut segment of $\mathcal{C}$ can be classified as either violating the point constraint or respecting it, independently of the whole simplification or other shortcut segments. Thus, if every shortcut segment of a simplification respects the point constraint, then the simplified curve must respect the point constraint also.

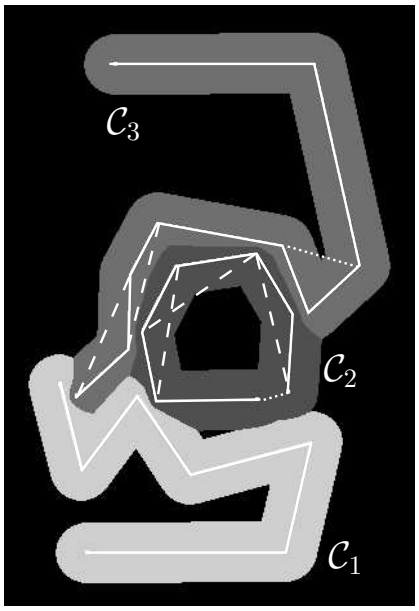


Figure 10: Step 3: Set of compliant proximate shortcut segments (dashed) and compliant nonproximate segments (dotted).

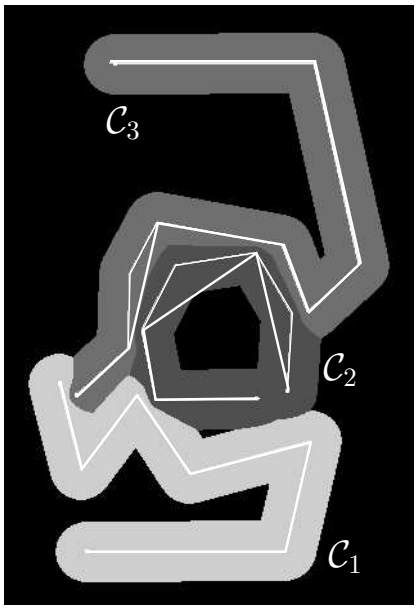


Figure 11: Step 4: Simplified chains (bold) and Input chains.

Recall that the algorithm starts by building a set of candidate shortcut segment edges. The point constraints can be incorporated in our algorithm simply by culling out all shortcut segments that violate point constraints. Given a shortcut segment $\overline{v_s v_e}$, Algorithm 5.2 shows how to check if $\overline{v_s v_e}$ respects point constraint for point p in time $O(|e - s + 1|)$.

Start with the shortcut segment $\overline{v_s v_{s+1}}$. Since it is part of the original chain $\mathcal{C}$, it cannot violate the point constraint. Then, the shortcut segment $\overline{v_s v_{s+2}}$ violates the point constraint if and only if the triangle formed by vertices v_s, v_{s+1} and v_{s+2} contains p. Applying induction, the two cases follow:

- **Triangle $v_s v_{s+i-1} v_{s+i}$ contains p** $\implies$ $\overline{v_s v_{s+i}}$ violates the point constraint p if and only if $\overline{v_s v_{s+i-1}}$ respects the point constraint p.

Algorithm 5.2 `Check_Point_Constraint`(Input Chain $\mathcal{C}_i$, Segment $\overline{v_s v_e}$, Point p)
Input: A Chain $\mathcal{C}_i = \{v_1, \ldots, v_{m_i}\}$, Segment $\overline{v_s v_e}$ and Constraint p
Output: '1' if $\overline{v_s v_e}$ respects p, '0' otherwise.

$p(\overline{v_s v_{s+1}}) = 1$ /* Initially set $p(\cdot)$ to respect segment $\overline{v_s v_{s+1}}$ */

for $i = 2$ to $(e - s)$ **do** { /* Iterate over all segments */ }
  if point p contained in triangle $v_s v_{s+i-1} v_{s+i}$ **then**
    $p(\overline{v_s v_{s+i}}) = 1 - p(\overline{v_s v_{s+i-1}})$ /* Need to flip validity */
  else
    $p(\overline{v_s v_{s+i}}) = p(\overline{v_s v_{s+i-1}})$
  end if
end for
Return $p(\overline{v_s v_e})$

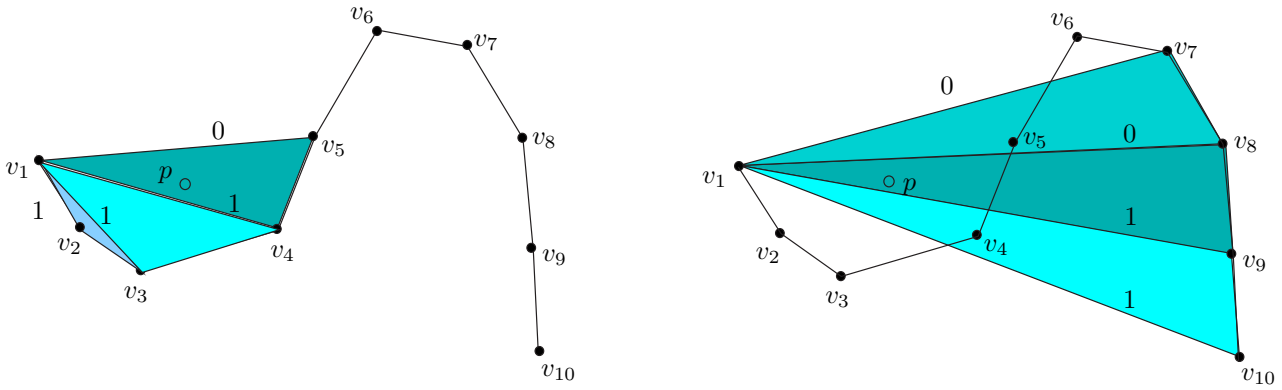


Figure 12: Checking for point constraints for segment $\overline{v_1 v_{10}}$. Labels of edges indicate the value of $p(\cdot)$.

- **Triangle $v_s v_{s+i-1} v_{s+i}$ does not contain p** $\implies$ $\overline{v_s v_{s+i}}$ violates the point constraint p if and only if $\overline{v_s v_{s+i-1}}$ violates the point constraint p.

Therefore, we can find for each segment $\overline{v_s v_{s+i}}$ whether it violates the point constraint or not incrementally. Performing the test of whether p is contained inside the triangle takes constant time, and therefore each segment $\overline{v_s v_{s+i}}$ can be checked in constant time, and the total time becomes $O(|e - s + 1|)$. See Fig. 12 for an illustration. The above algorithm can be extended to address multiple point constraints — a shortcut segment respects all the point constraints if it respects the point constraint for each constraint point p.

6 Interactive Visualization

The above algorithm can be used either as a stand-alone map simplification technique or as part of an interactive system that simplifies maps in real-time based on the current viewpoint. However, for large maps that run into hundreds of millions of vertices, the above approach is unsuitable, because it requires us to maintain the entire map in memory at all times. In this section, we present an interactive out-of-core system for visualizing large topographic maps. It uses prefetching and a level-of-detail data structure to manage large disk-resident maps and handle simplification requests efficiently. The system uses two processes, one for rendering and the second one for I/O management. Since the rendering pipeline and the I/O pipeline are independent of each other, both of these can run in parallel.

In addition to performing geometric simplification, our system also makes use of *semantic* level-of-detail information. In contrast to map generalization systems that study the problem of automatic region aggregation, we assume that the semantic aggregation levels are provided to us in advance. The system works independently of the techniques used for geometric simplification and aggregation, and thus is of independent interest.

6.1 Data Representation

6.1.1 Data Organization

Our input data consists of more than 50,000 maps from the TIGER [5] database, with a total of over 100 million vertices. The database occupies more than 1 GB on disk. The maps are organized in a tree (the *Data Tree*) as shown in Fig 13.

Each level of the data tree represents a different semantic level of detail (LOD). Maps at a lower level of the data tree are at a higher level of detail. For example, Fig 14 shows the map of New Jersey at two different semantic levels of detail, one at a county level and the second at a block level.

As part of preprocessing, we generate simplifications of each of the original maps. We refer to these simplifications as geometric levels of detail (LOD) (Fig 13). In addition, we also compute bounding boxes for each node of *Data*

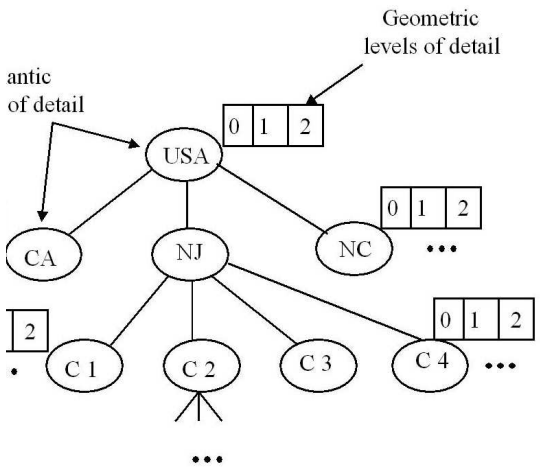


Figure 13: *Data Tree*: Root node corresponds to USA, its children correspond to different states, grandchildren correspond to counties and so on. The height of our *Data Tree* is 5. Each node is associated with a geometric level of detail. For example, the root node has three geometric levels of detail; index 0 corresponds to the finest unsimplified map, index 2 corresponds to the coarsest simplified map.

Algorithm 6.1 Traverse(node)
Input: Node of *Data Tree*
Output: Map at appropriate level of geometric and semantic detail.

```
if node's bounding box is outside the view frustum then
   return
end if
if area of projection of node's bounding box < threshold then
   Of all maps M[i] associated with the the node N,
   choose one with largest i such that
   ε M[i] <ε                    /*where ε is the user-specified threshold. */
else
   for each child node cnode do
      Traverse (cnode)
   end for
end if
```

Tree. Each simplified map M is associated with an error measure E_M. At runtime, we project a ball of radius E_M placed at the centroid of the map onto the screen. If the area of the projection (ϵ_{M}) is less than a user-specified threshold, then the simplified map can be selected.

6.1.2 Data Tree Traversal

In order to render the maps, we begin traversing the *Data Tree* at the root node (see Algorithm 6.1). We test if any given node's bounding box is outside the view frustum and in that case we stop traversing further. We then check if the current node can be chosen as a semantic LOD based on whether the area of projection of its bounding box onto the screen is less than a threshold. If the current node is chosen as the semantic LOD, then we choose the coarsest map that satisfies the screen-space error constraint as the geometric level of detail. Otherwise, we recurse on each of the node's children.

At the end of the traversal of the *Data Tree*, we have a set of maps that were chosen and need to be rendered. We refer to this set of maps as the *front*.

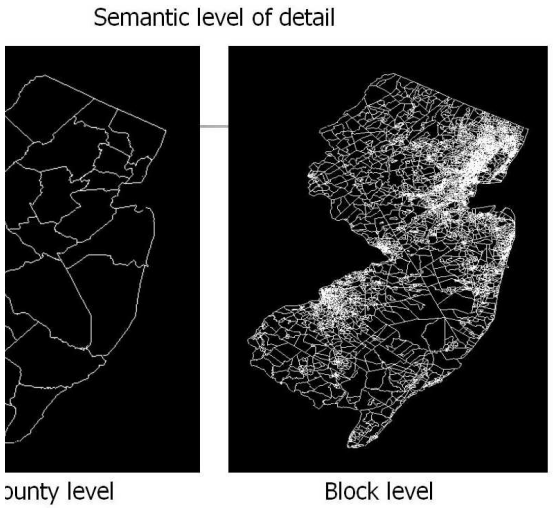


Figure 14: Semantic LOD: This figure shows two different semantic levels of detail for New Jersey state (NJ). The figure on the left shows New Jersey at a county level, the one on the right at a block level.

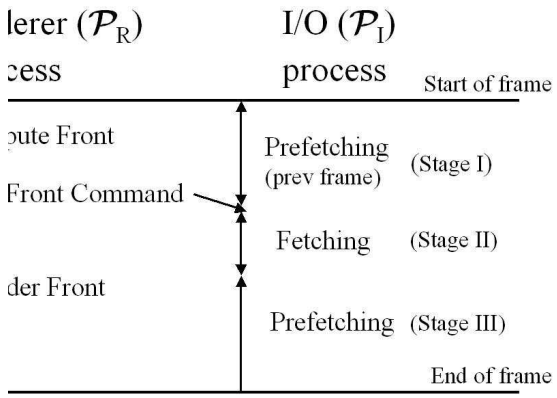


Figure 15: Parallel Processes: Our algorithm uses two processes, one for rendering and one for I/O. The figure shows the tasks performed by each of the two processes in a given frame time.

6.1.3 Out-of-core representation

In order to traverse the *Data Tree* and compute the front, our algorithm only needs the *Data Tree* and the skeleton. The skeleton includes the nodes and connectivity information like parent-child relationships, as well as additional data structures including bounding boxes and error metrics associated with the simplified maps. It typically takes only a small fraction of the overall data representation. This division of the model into in-core and out-of-core representations ensures that the main memory overhead is almost equal to the size of the skeleton. The rendering algorithm accepts a memory footprint size as input and we ensure that its memory usage cannot exceed this limit. Moreover, we assume that the given memory footprint is large enough to hold the skeleton.

6.2 Out-of-Core Rendering

6.2.1 Parallel Rendering & I/O

Our algorithm uses two main processes: one for *Data Tree* traversal and rendering ($\mathcal{P}_\mathcal{R}$), and the second one for I/O management and prefetching ($\mathcal{P}_\mathcal{I}$). Both of them run in parallel and operate synchronously (see Fig 15). $\mathcal{P}_\mathcal{R}$ traverses the *Data Tree* and computes the front based on the current viewpoint and *Data Tree* skeleton. Once $\mathcal{P}_\mathcal{R}$ finishes the front computation, it sends a fetch command to $\mathcal{P}_\mathcal{I}$. On receiving a fetch command, $\mathcal{P}_\mathcal{I}$ gets synchronized

with $\mathcal{P}_{\mathcal{R}}$ and begins loading maps that belong to the front but are not in memory (Stage II). Different maps that constitute the front can be rendered in any order. As a result, $\mathcal{P}_{\mathcal{R}}$ starts rendering the maps that are in memory and does not wait till all the maps are loaded. The rendering and the loading of out-of-core maps proceeds in parallel. If $\mathcal{P}_{\mathcal{R}}$ has rendered all the maps in memory, it has to wait till new maps are loaded. Once $\mathcal{P}_{\mathcal{I}}$ has fetched all the maps that belong to the front but are not in memory, it spends time prefetching other maps that may be needed for subsequent frames (Stage III & Stage I of next frame).

6.2.2 Prefetching

As the viewpoint moves, the front changes in many ways. These include different events.

- Semantic LOD Switching Events.
  A map that was in the front may be replaced either by a map from a parent node or by a set of maps from a child node in the *Data Tree*. To avoid misses due to semantic LOD switches, for each map in the front, we could prefetch maps from its ascendant and descendant nodes.

- Geometric LOD Switching Events.
  A map that was in the front may be replaced by a coarser or finer map associated with the same node. We could prefetch coarser and finer maps associated with the same node.

- Visibility Events.
  A map that wasn't (resp. was) in the front may appear (resp. disappear) because the corresponding node is visible (resp. not visible). Typically in case of view frustum culling, these maps are outside the view frustum but close to the edge of the view frustum. We can make the prefetching view frustum larger than the rendering view frustum and avoid misses due to visibility events.

To handle large fronts, we assign priorities based on closeness to thresholds in a manner similar to the one described in [36]. Given an upper bound on the size of main memory, we need a mechanism to remove or replace some of the maps from the main memory. We use the replacement policy described in [36].

7 Performance Analysis

In this section we report the results of experiments testing the performance and quality of our system.

7.1 Map Simplification

To demonstrate the quality of simplification produced by our method, we ran the algorithm on three different maps. The first map (called USA) is a map of the United States that contains the national and state boundaries. It has 261,460 vertices and 375 chains. Figs. 17,18 illustrate the output from our map simplification method, for varying values of the tolerance parameter ε.

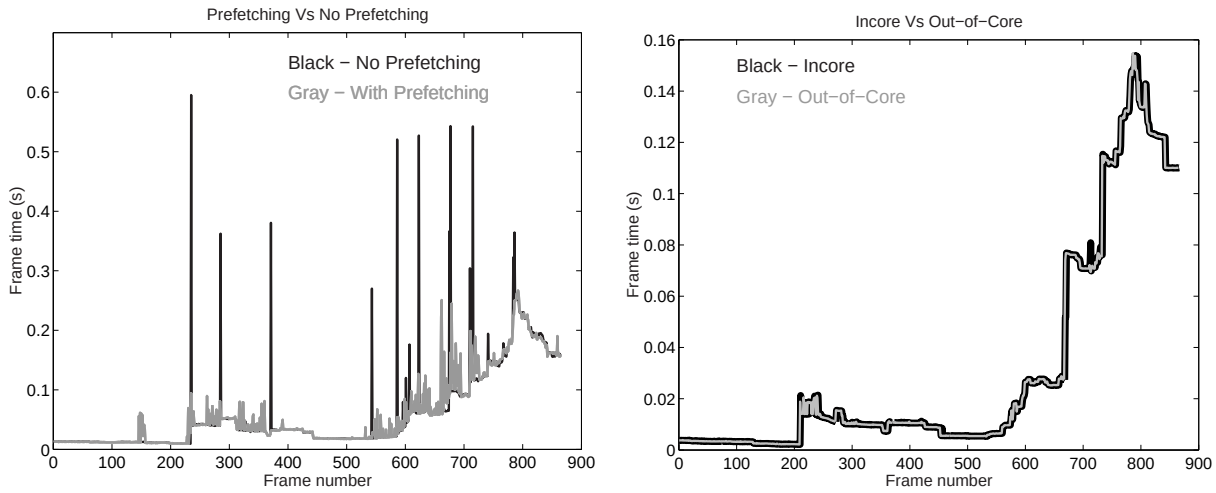
The second map is a map of Asia that contains both national and some regional boundaries. It has 233,320 vertices and 1,164 chains. Figs. 25,26 illustrate the different simplified levels.

The third map is a contour map of California, generated using land elevation data from the National Geophysical Data Center [1]. The statistical software Splus [2] was used to generate the contour map. This map has 357,350 vertices and 18,515 chains. Fig. 19 shows the original map, and Figs. 20(a), 21(a) show zoomed-in regions of the map, the corresponding simplified regions being shown in Figs. 20(b), 21(b). This map was simplified with ε=0.002, and the simplified map contains 73,605 vertices.

7.2 Point Constraints

We demonstrate the manner in which our algorithm deals with point constraints with an example from the state of Massachusetts. Fig. 22 displays a portion of the state boundary of Western Massachusetts with four coastal points marked in white dots. In Fig. 23 we display the regions marked in circles more closely. Notice that in the original map, two of the three points are on land, and one is on water. Fig. 24(a) illustrates the simplification obtained using our point constraint algorithm. The sidedness of the points is preserved, even though there is significant simplification. On the other hand, in Fig. 24(b) we see the resulting map when point constraints are not taken into account: two of the three points have changed their side.

7.3 Interactivity Results



(a) Prefetching Vs No Prefetching

(b) Incore Vs Out-of-Core

Figure 16: Frame time: Plots in black & gray in the figure on the left compare the frame times for the out-of-core system without prefetching (*OOCNoPre*), & out-of-core system with prefetching (*OOCPre*), respectively on *Machine 1* that has 128 MB of main memory. Plots in black & gray in the figure on the right compare the frame times for *OOCPre* & the in-core system (*Incore*) respectively on *Machine 2* that has 16 GB of main memory. The *Incore* system needed more than 1 giga-byte of main memory for rendering. The *OOCNoPre* and *OOCPre* systems used a memory footprint of 25 MB. We see that the *OOCPre* system matches the performance of the *Incore* system.

We implemented our out-of-core rendering algorithm and tested it on two machines: an SGI workstation with two 195 MHZ R10000 MIPS processors, MXI graphics board, 128 MB of main memory (*Machine 1*) and an SGI Onyx with multiple 500 MHZ R14000 MIPS processors, Infinite Reality3 graphics pipelines and 16 GB of main memory (*Machine 2*). We performed tests on *Machine 1* in order to show that our out-of-core algorithms require a limited memory footprint and on *Machine 2* to compare the performance of our out-of-core rendering algorithm with that of an in-core rendering algorithm. Fig 16 show the frame time plots comparing three systems:

- *Incore*: The in-core system that loads the entire *Data Tree* and all the maps in the main memory.
- *OOCNoPre*: Initially loads the *Data Tree* skeleton and performs parallel rendering & I/O management without any prefetching. It fetches data from the disk in Stage II, but is idle during Stages I & III (see Fig 15).
- *OOCPre*: Initially loads the skeleton and performs both parallel rendering & I/O management as well as prefetching (*OOCPre*).

The overall performance of the *OOCNoPre* system matches with that of the *OOCPre* system at many places in the sample paths. However, the frame time plot of *OOCNoPre* system has several spikes, which typically correspond to relatively large changes in the viewpoint resulting in drastic changes in the front. The maps in the new front may not have been in the main memory and this can result in more misses, which increases the fetching time. The overall rendering or frame time is not affected as long as the time to fetch maps that are not in memory is less than the time to render maps that are in memory. This is the main benefit of performing rendering & I/O management in parallel. However, a large number of misses can cause the fetching time to dominate the rendering time and this results in spikes in the frame time plot. The memory usage is equal to the amount of data stored in main memory at any given time.

Fig 16 compares the performance of *OOCPre* with *OOCNoPre* and *Incore*. They highlight the benefits of prefetching. We see that the frame time plot for *OOCPre* does not have any spikes. Its frame rate does not have major variations due to the I/O bottleneck. Moreover, the performance of *OOCPre* matches that of the in-core system, *Incore*. Also the memory usage of *OOCPre* does not exceed the memory footprint limit.

8 Discussion

In this paper, we describe an interactive system for visualizing large maps that has at its core a fast map simplification algorithm. Our method is based on the exploitation of modern graphics hardware to perform key computations efficiently, and is fast, flexible, and simple to implement.

One important issue that we fail to address in this paper is the problem of self-intersections. Although our techniques will ensure that no two chains intersect, it is possible that in the process of simplifying, a chain might intersect itself. Eliminating self-intersections while simultaneously attempting to minimize the length of the simplified chain is a hard problem; a recent attempt to address this problem is the work of Mantler and Snoeyink [27], where they define *safe sets* as a way of guaranteeing that standard simplification algorithms will not create intersections. It is possible that their techniques might be applicable to our method.

The paradigm of simulating general purpose computations on the graphics pipeline is a recent trend in graphics and visualization [25, 15, 21, 31]. Especially in the domain of geometric computation, there is the hope of obtaining fast algorithms for a variety of problems using techniques similar to those that we used. Some problems that seem amenable to such methods are mesh simplification, medial axis computation, cartogram computation and others. This is a direction that we plan to pursue actively in the future.

9 Acknowledgments

We would like to thank Jack Snoeyink and Andrea Mantler for helpful comments. Allan Wilks and Eleftherios Koutsofios helped us to get the map data that we used in our experiments. Numerous helpful suggestions from the anonymous referees helped us to improve our presentation and correct errors.

References

[1] National geophysical data center. `http://www.ngdc.noaa.gov/`.

[2] Splus. `http://www.insightful.com`.

[3] AGARWAL, P., KRISHNAN, S., MUSTAFA, N., AND VENKATASUBRAMANIAN, S. Streaming geometric optimization using graphics hardware. In *Proc. 11th European Symposium on Algorithms* (2003).

[4] AGARWAL, P. K., HAR-PELED, S., MUSTAFA, N., AND WANG, Y. Near-linear time approximation algorithms for curve simplification in two and three dimensions. In *Proc. of the 10th European Symposium on Algorithms* (2002), pp. 29–41.

[5] BUREAU, U. S. C. TIGER: Topologically integrated geographic encoding and referencing system. `http://www.census.gov/geo/www/tiger/`.

[6] CHAN, W. S., AND CHIN, F. Approximation of polygonal curves with minimum number of line segments or minimum error. *Internat. J. Comput. Geom. Appl. 6*, 1 (1996), 59–77.

[7] COHEN, J., VARSHNEY, A., MANOCHA, D., TURK, G., WEBER, H., AGARWAL, P., BROOKS, JR., F. P., AND WRIGHT, W. Simplification envelopes. In *Proc. 23th ACM SIGGRAPH* (1996), pp. 119–128.

[8] DE BERG, M., VAN KREVELD, M., AND SCHIRRA, S. A new approach to subdivision simplification. In *ACMS/ASPRS Annual Convention and Exposition* (1995), vol. 4, pp. 79–88.

[9] DOUGLAS, D. H., AND PEUCKER, T. K. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *The Canadian Cartographer 10* (1973), 112–122.

[10] EDELSBRUNNER, H. *Algorithms in Combinatorial Geometry*, vol. 10 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Heidelberg, West Germany, 1987.

[11] ESTKOWSKI, R. No Steiner point subdivision simplification is NP-Complete. In *Proc. 10th Canadian Conf. Computational Geometry* (1998).

[12] ESTKOWSKI, R., AND MITCHELL, J. S. B. Simplifying a polygonal subdivision while keeping it simple. In *17th ACM Symposium on Computational Geometry* (2001), pp. 40–49.

[13] GARLAND, M., AND HECKBERT, P. Surface simplification using quadric error bounds. *Proc. 24th ACM SIGGRAPH* (1997), 209–216.

[14] GOLDFEATHER, J., MOLNAR, S., TURK, G., AND FUCHS, H. Near real-time CSG rendering using tree normalization and geometric pruning. *IEEE Computer Graphics and Applications 9*, 3 (May 1989), 20–28.

[15] GREENE, N., KASS, M., AND MILLER, G. Hierarchical Z-buffer visibility. In *Computer Graphics (SIGGRAPH '93 Proceedings)* (aug 1993), pp. 231–238.

[16] GUESGEN, H. W., HERTZBERG, J., LOBB, R., AND MANTLER, A. First steps towards buffering fuzzy maps with graphics hardware. In *Proc. FOIS Workshop on Spatial Vagueness, Uncertainty and Granularity (SVUG-01)* (2001).

[17] GUIBAS, L. J., HERSHBERGER, J. E., MITCHELL, J. S. B., AND SNOEYINK, J. S. Approximating polygons and subdivisions with minimum link paths. *Internat. J. Comput. Geom. Appl. 3*, 4 (1993), 383–415.

[18] HALPERIN, E., NATHANIEL, R., AND ZWICK, U. Coloring k-colorable graphs using smaller palettes. In *Proc. 12th ACM-SIAM Symposium on Discrete Algorithms* (2001), pp. 319–326.

[19] HERSHBERGER, J., AND SNOEYINK, J. Speeding up the Douglas-Peucker line simplification algorithm. In *Proc. 5th Internat. Sympos. Spatial Data Handling* (1992), pp. 134–143.

[20] HERSHBERGER, J., AND SNOEYINK, J. Computing minimum length paths of a given homotopy class. *Comput. Geom. Theory Appl. 4* (1994), 63–98.

[21] HOFF III, K. E., KEYSER, J., LIN, M., MANOCHA, D., AND CULVER, T. Fast computation of generalized Voronoi diagrams using graphics hardware. In *Computer Graphics (SIGGRAPH '99 Proceedings)* (1999), vol. 33, pp. 277–286.

[22] HOPPE, H. Progressive meshes. In *Computer Graphics (SIGGRAPH '96 Proceedings)* (1996), vol. 30, pp. 99–108.

[23] HOPPE, H., DEROSE, T., DUCHAMP, T., MCDONALD, J., AND STUETZLE, W. Mesh optimization. In *Computer Graphics (SIGGRAPH '93 Proceedings)* (Aug. 1993), vol. 27, pp. 19–26.

[24] KRISHNAN, S., MUSTAFA, N., AND VENKATASUBRAMANIAN, S. Hardware-assisted depth contours. In *Proc. 13th ACM-SIAM Symposium on Discrete Algorithms* (2002), pp. 558–567.

[25] KRISHNAN, S., SILVA, C. T., AND WEI, B. Hardware-assisted visibility-ordering algorithm with applications to volume rendering. In *Data Visualization* (2001), pp. 233–242.

[26] LINDSTROM, P., AND TURK, G. Image-driven simplification. *ACM Transactions on Graphics 19*, 3 (2000), 204–241.

[27] MANTLER, A., AND SNOEYINK, J. Safe sets for line simplification. In *10th Annual Fall workshop on Computational Geometry* (2000).

[28] MARINO, J. S. Identification of characteristic points along naturally occurring lines / an empirical study. *The Canadian Cartographer 16*, 1 (1979), 70–80.

[29] MCMASTER, R. Automated line generalization. *Cartographica 24*, 2 (1987), 74–111.

[30] MCMASTER, R., AND SHEA, K. *Generalization in Digital Cartography*. Association of American Geographers, 1992.

[31] PEERCY, M. S., OLANO, M., AIREY, J., AND UNGAR, P. J. Interactive multi-pass programmable shading. In *Proc. 27th ACM SIGGRAPH* (2000), pp. 425–432.

[32] POPOVIC, J., AND HOPPE, H. Progressive simplicial complexes. In *Proc. ACM SIGGRAPH* (1997), pp. 217–224.

[33] ROSSIGNAC, J., AND BORREL, P. Multi-resolution 3D approximation for rendering complex scenes. In *Second Conference on Geometric Modelling in Computer Graphics* (June 1993), pp. 453–465.

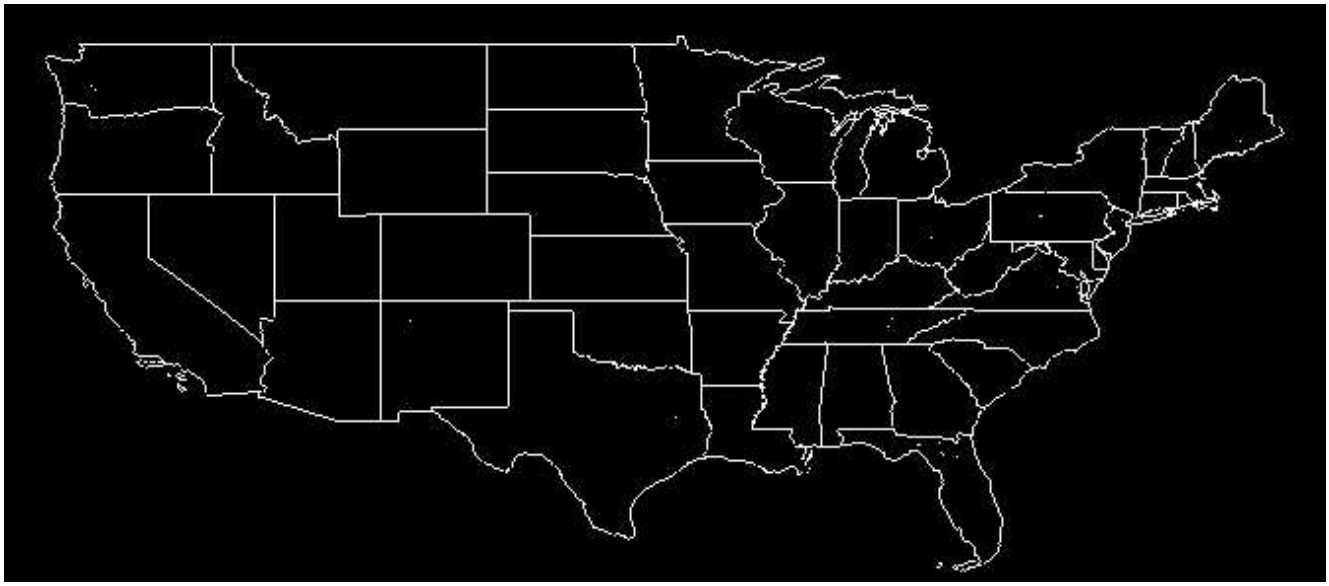
[34] ROSSIGNAC, J. R., AND REQUICHA, A. A. G. Depth-buffering display techniques for constructive solid geometry. *IEEE Computer Graphics and Applications 6*, 9 (1986), 29–39.

[35] SCHROEDER, W. J., ZARGE, J. A., AND LORENSEN, W. E. Decimation of triangle meshes. In *Computer Graphics (SIGGRAPH '92 Proceedings)* (1992), vol. 26, pp. 65–70.

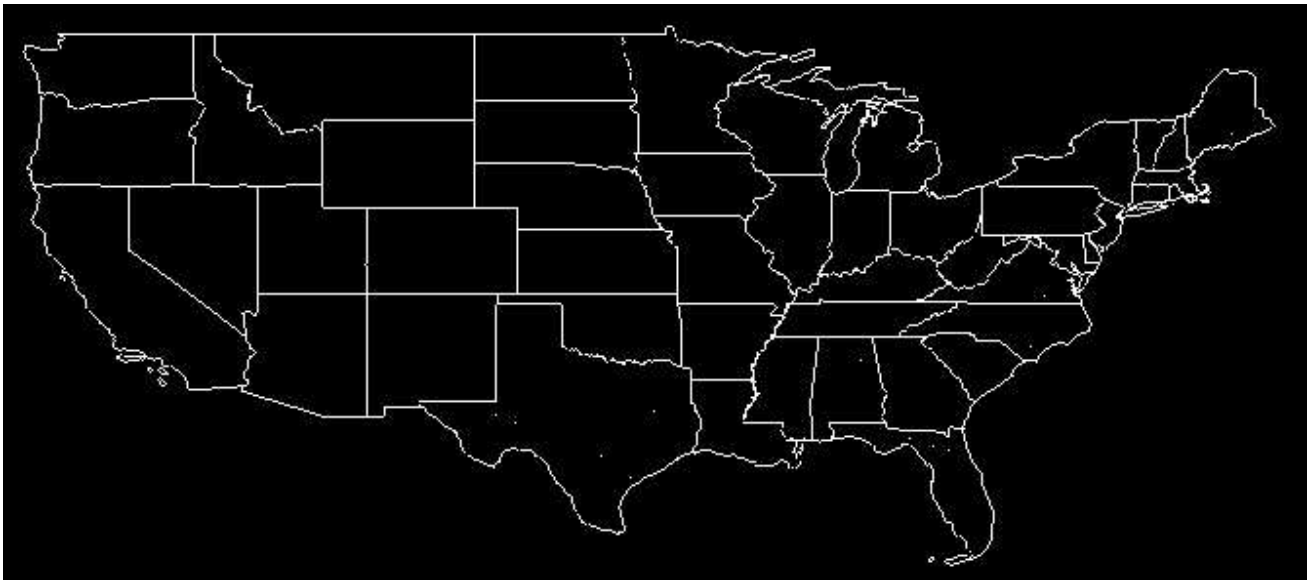
[36] VARADHAN, G., AND MANOCHA, D. Out-of-core rendering of massive geometric datasets. In *IEEE Visualization* (2002), pp. 69–76.

[37] WEIBEL, R., AND JONES, C. Computational perspectives on map generalization. *GeoInformatica 2*, 4 (1998), 307–314.

[38] WHITE, E. R. Assessment of line-generalization algorithms using characteristic points. *The American Cartographer 12*, 1 (1985), 17–27.

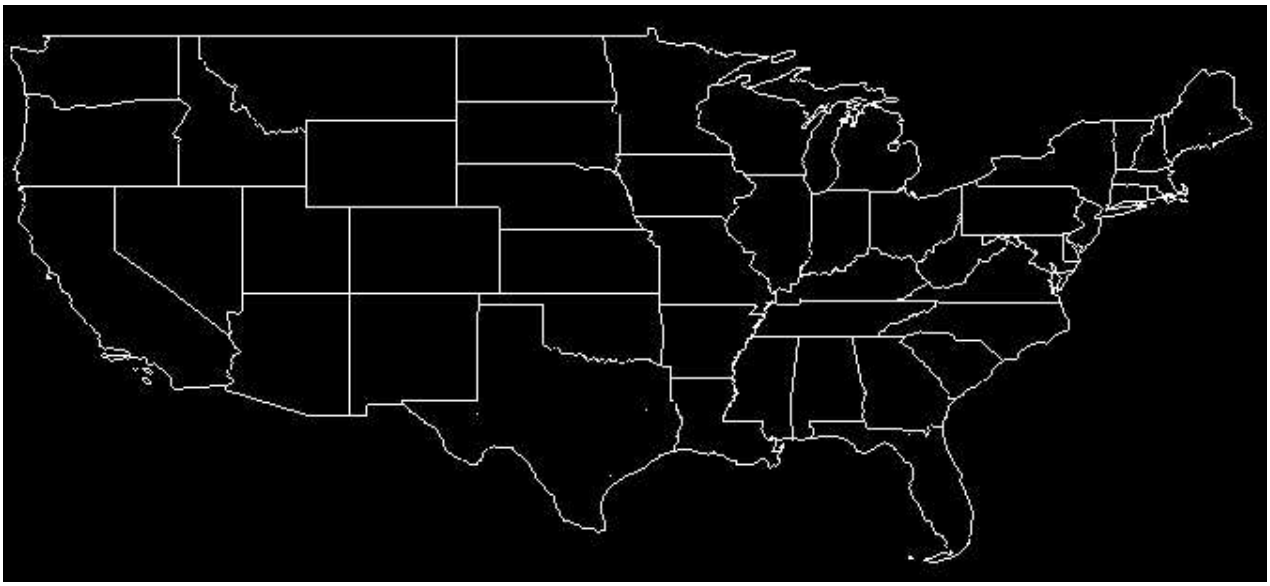


(a) The original map: 261,460 vertices

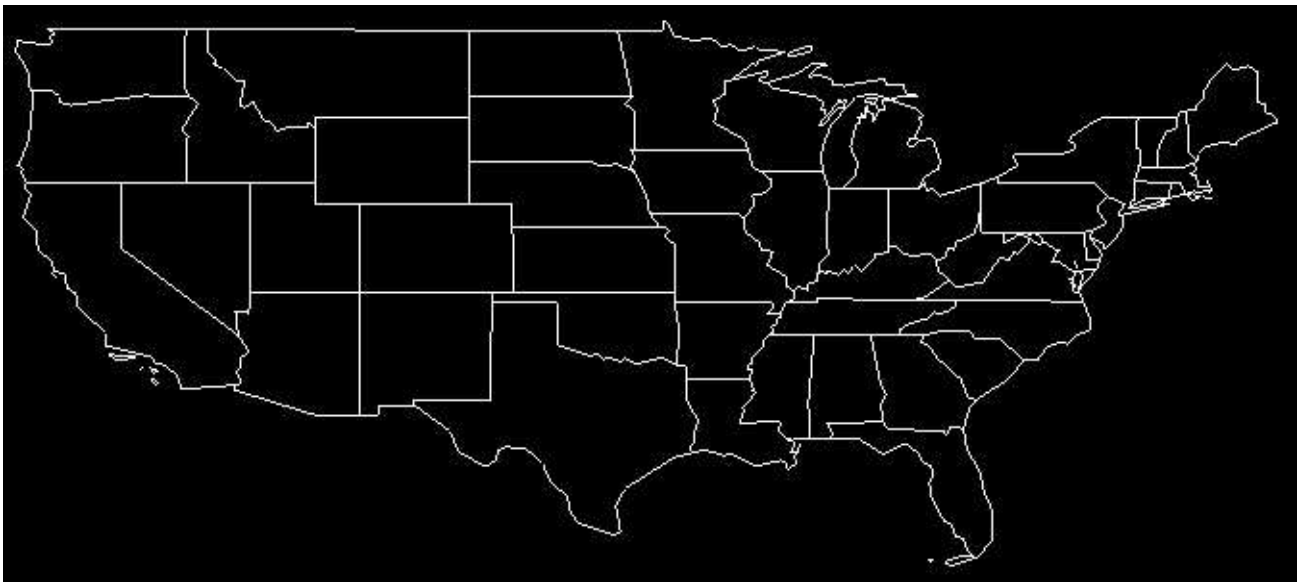


(b) 9,718 vertices, ε =0.0002

Figure 17: Map of the United States at different levels of detail.



(a) 3,287 vertices, ε =0.0008



(b) 1,233 vertices, ε =0.004

Figure 18: Map of the United States at different levels of detail (cont.).

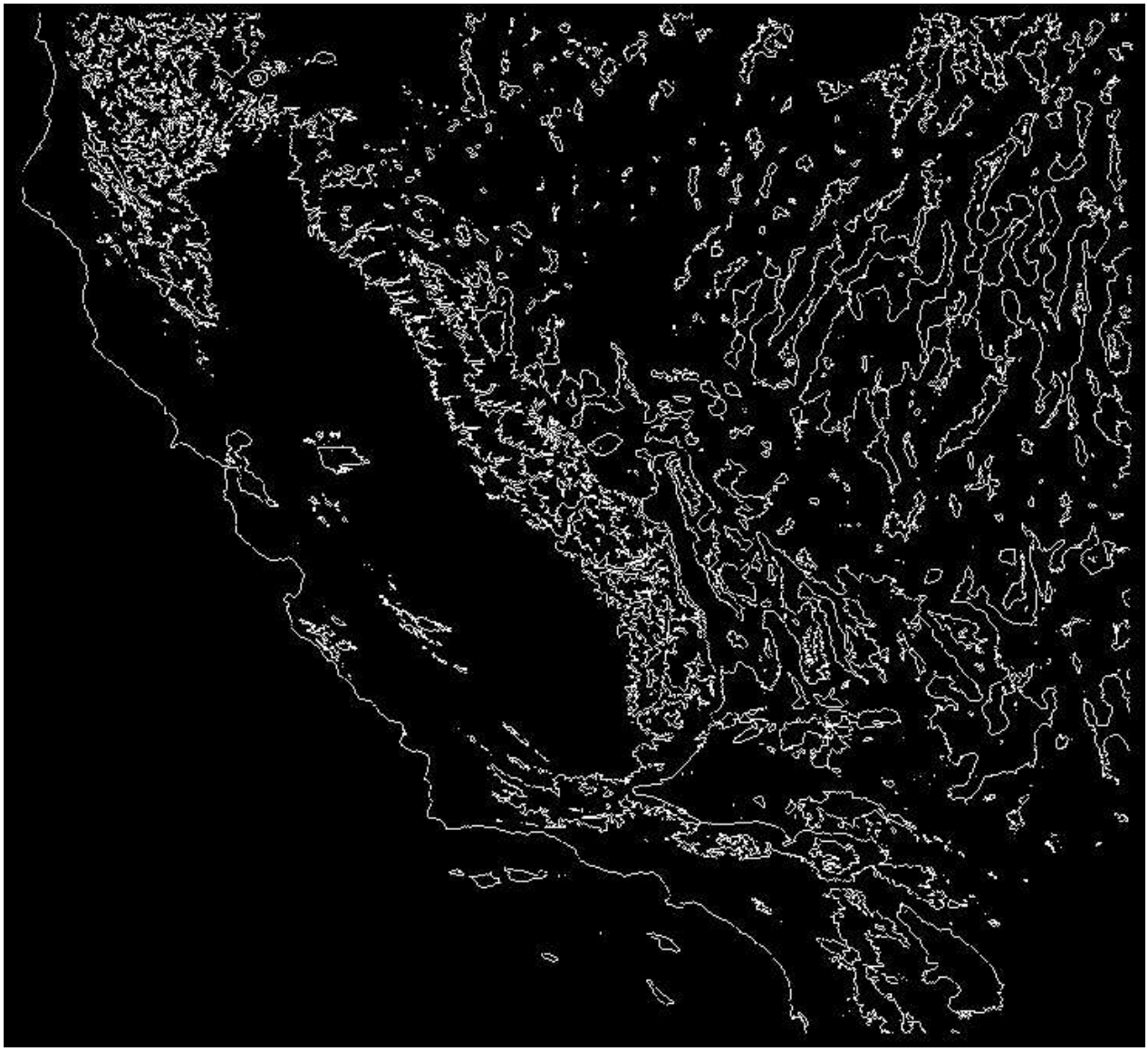


Figure 19: Contour Map of California: 357,350 vertices and 18,515 chains.

(a) Contours of the Sierra Nevada Range.

(b) Simplified contours of the Sierra Nevada Range.

Figure 20:

(a) Contours of the Cascade Range.

(b) Simplified contours of the Cascade Range.

Figure 21:

Figure 22: A map of the boundary of Western Massachusetts with four coastal points marked.

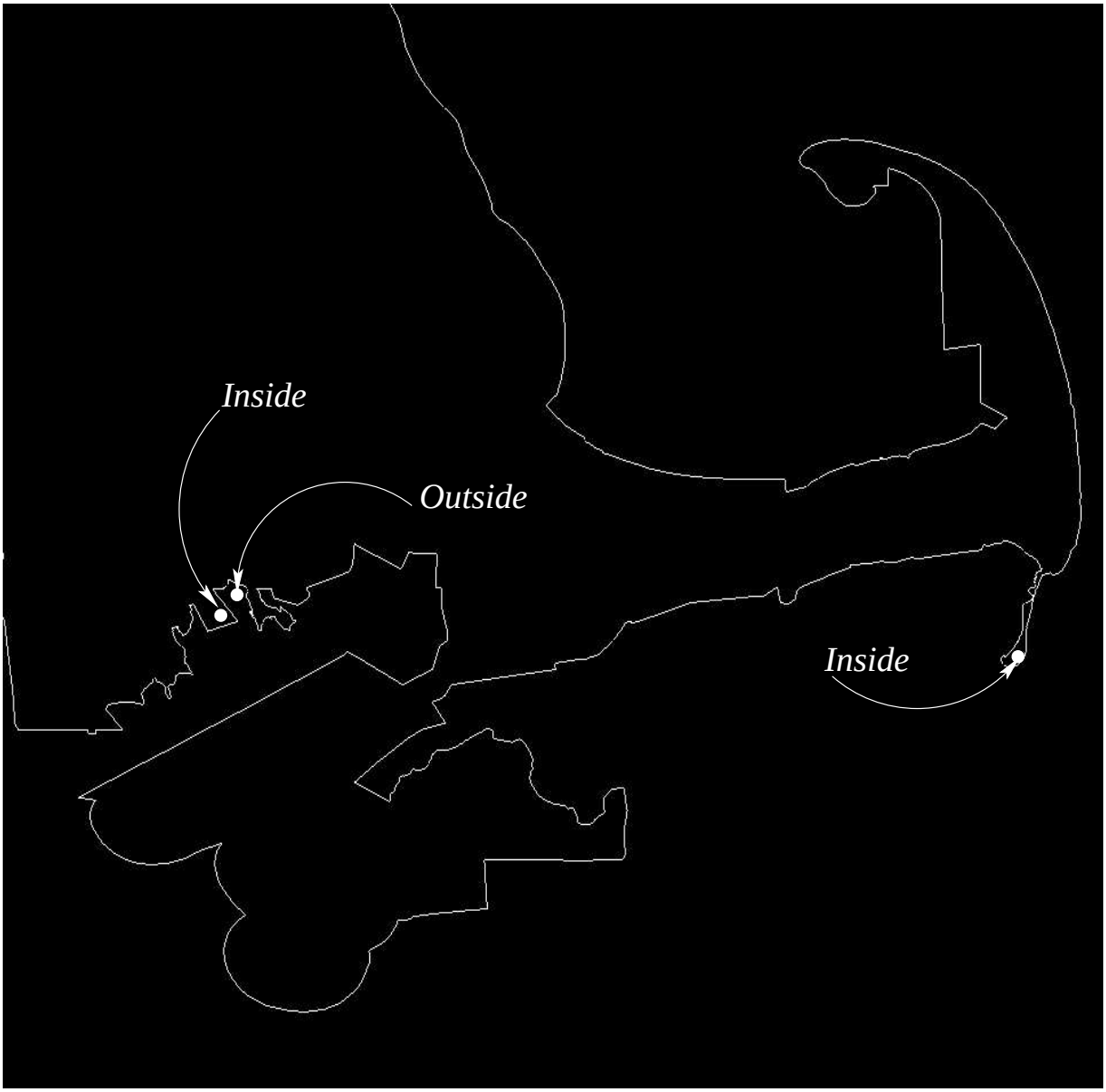
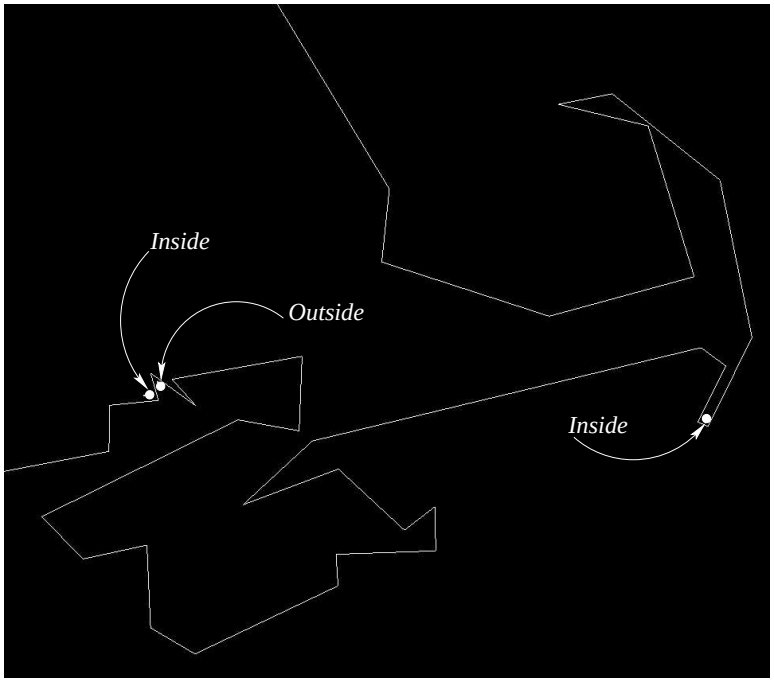
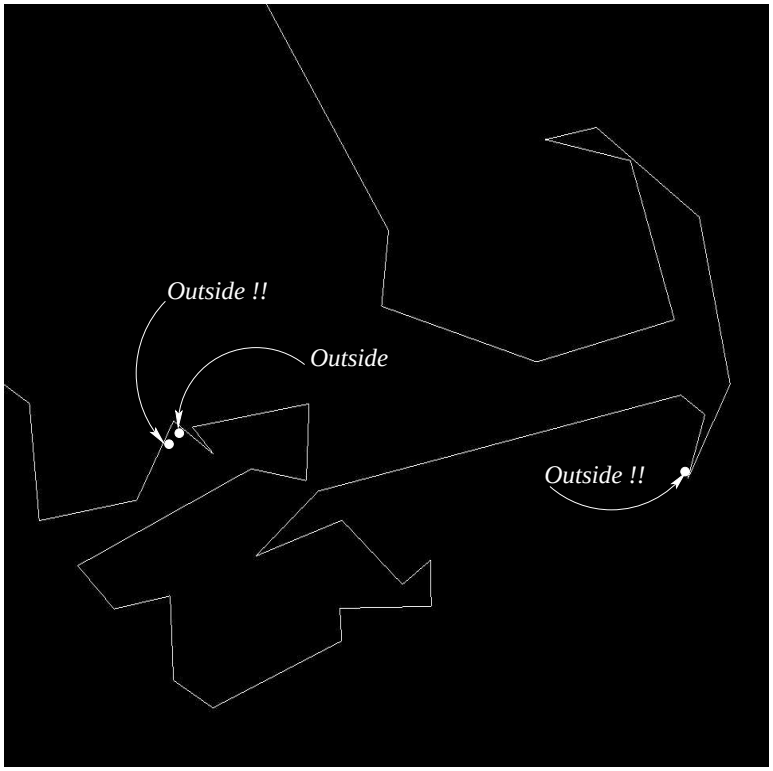


Figure 23: A zoomed-in view of three of the coastal points.

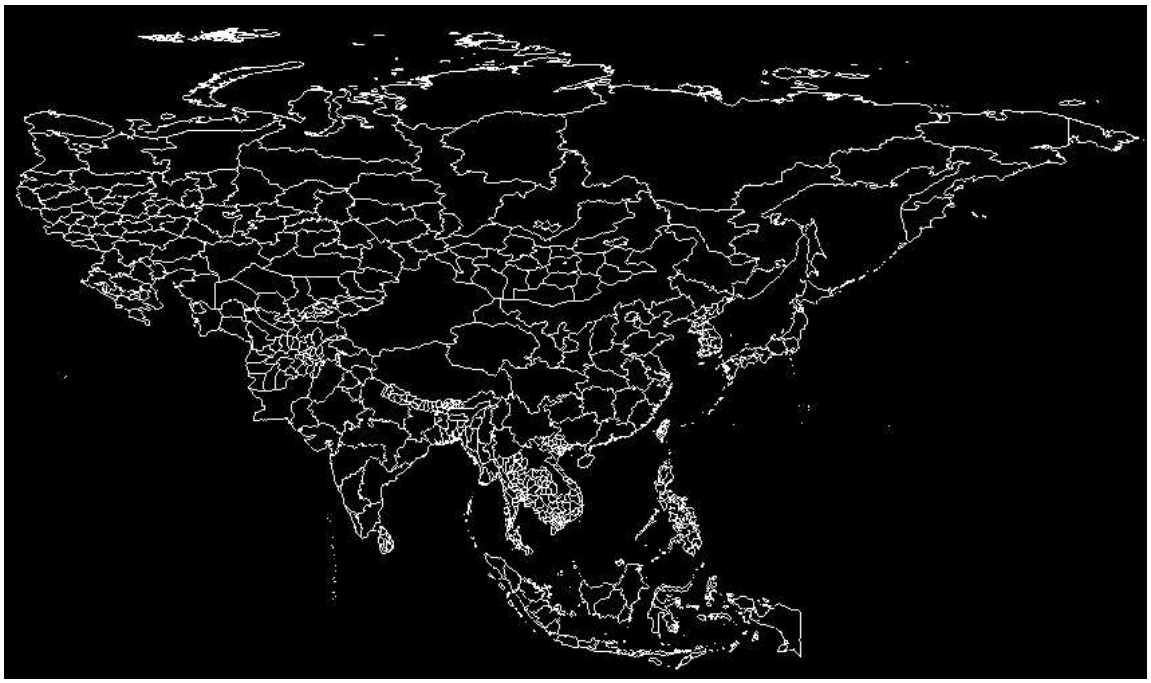


(a) Simplification produced by the algorithm when point constraints are considered. The three points are on the correct sides of the map.

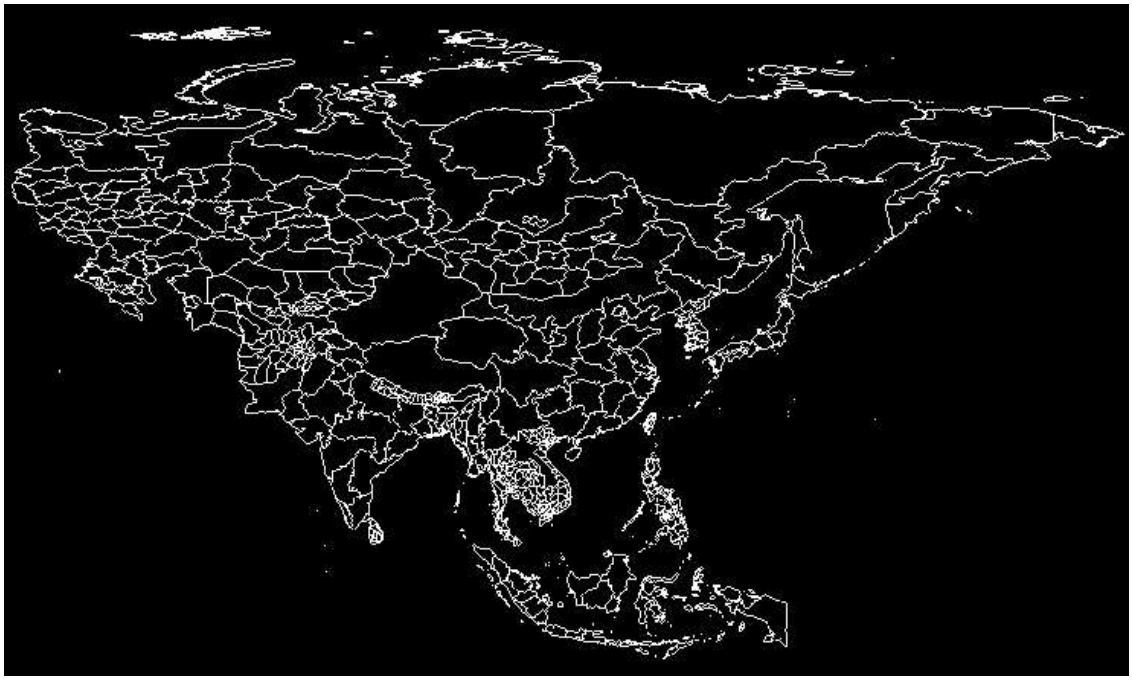


(b) Simplification produced when point constraints are not considered, Two points have changed their side.

Figure 24: The use of point constraints.

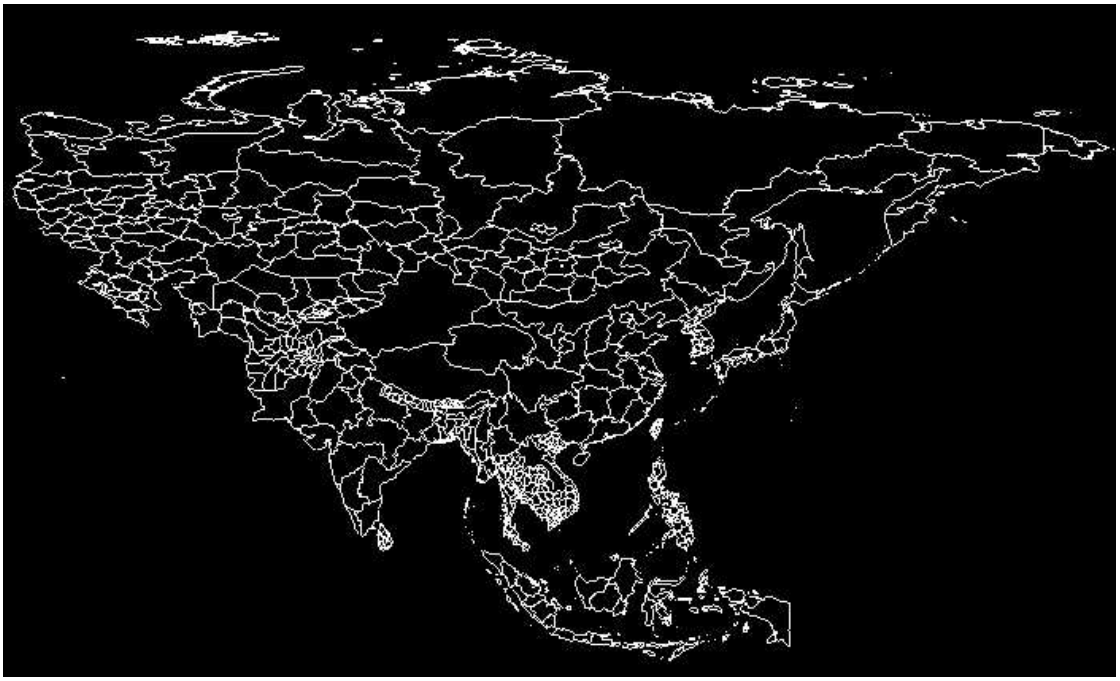


(a) The original map: 233,320 vertices

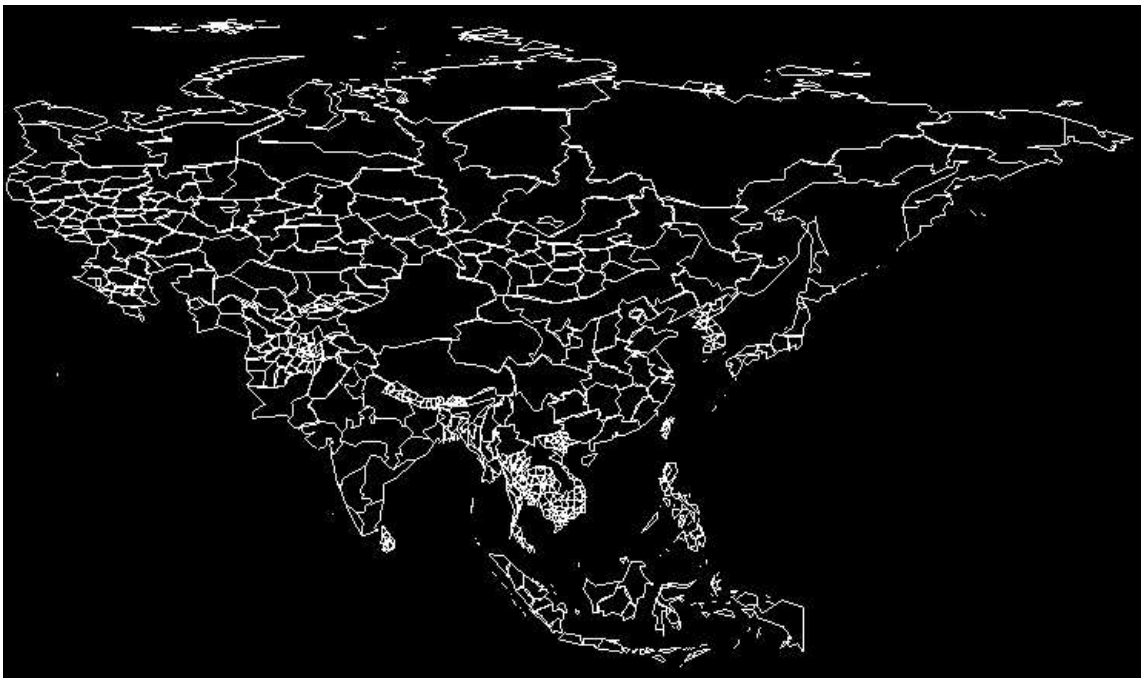


(b) 49,445 vertices, $\varepsilon = 0.0002$

Figure 25: Map of Asia at different levels of detail.



(a) 20,710 vertices, $\varepsilon = 0.0008$



(b) 6,744 vertices, $\varepsilon = 0.004$

Figure 26: Map of Asia at different levels of detail (cont.).