Hardware-Assisted View-Dependent Planar Map Simplification

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

Eleftheris Koutsofios
AT&T Labs – Research
ek@research.att.com

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

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

ABSTRACT

In this paper, we present an algorithm and a system to perform dynamic view dependent 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. We also present an interactive map visualization system which uses frame-to-frame coherence to perform dynamic view-dependent simplification. Our initial results indicate that we get a 3-4 fold increase in the frame rates using our simplification algorithm on maps with 1.5-2 million vertices on an SGI Onyx workstation.

1. INTRODUCTION

With the recent advances in satellite imaging techniques, the ability to acquire highly detailed geographical map models has become a reality. This is crucial for the interactive visual exploration of massive data sets, much of which is spatial in nature, and is generated with respect to an underlying geographic domain. For example, geographic information systems deal with the problem of capture, storage and display of large amounts of data with spatial information like weather or elevation information, while large telecommunications networks are visualized over an underlying global map [22]. Thus, one of the major challenges facing any interactive system for visualizing such data is the problem of rendering the underlying spatial domain (which could be a political or topographical map of a region, or even a three-dimensional terrain). Map data can be extremely complex in its finest detail; the Tiger [3] data provided by the U.S. Census Bureau contains U.S. maps with a few hundred million vertices occupying over 30 GB of storage.

The maps created by such 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. This is known in the graphics community as a *view-dependent* 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*. This is the problem of interactively maintaining a planar subdivision (arrangement of polylines) 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) of the region are preserved. In order to achieve real-time performance, we implement key steps of our simplification algorithm using the 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.

Main Contributions: The main features of our simplification algorithm are:

Use of graphics hardware: Most existing automated methods for map simplification (see Section 3) use software-based geometric methods that may not scale well and can have problems of robustness due to geometric degeneracy. We employ extensive use of the OpenGL graphics pipeline; this improves the performance of our system greatly, and provides an interesting avenue for further research into the implementation of geometric algorithms (see Section 8).

System implementation: We have implemented all parts of our algorithm and developed a system which performs a real-time visualization of maps. The system performs dynamic, view-dependent simplification to determine the primitives to be rendered at every frame. Our system provides 15-20 frames per second on maps containing up to 500,000 vertices.

Metric independence: In order to define the notion of shape

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

preservation, any simplification algorithm must assume that the object to be simplified is embedded in a metric space. However, our algorithm does not favor any particular distance function. This is important because notions of shape preservation can change depending on the application.

Constrained simplification: We can extend our algorithm to accommodate additional geometric 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.

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.

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. The details of our simplification algorithm are given in Section 5 and the issues that we faced in designing our interactive map visualization system and our solutions are mentioned in Section 6. 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 between a point v_i and a segment $\overline{v_j v_k}$ is defined by

$$d(v_i, \overline{v_j v_k}) = \min_{p \in \overline{v_j v_k}} d(v_i, p)$$

We define the $error$ of a line segment, $\overline{v_i v_j}$ 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 are a subsequence of the vertices of $\mathcal{C}$. Then we define the distance $d(\cdot,\cdot)$ between the two chains as

$$d(\mathcal{C}', \mathcal{C}) = \max_{1 \leq j < m'} \Delta(\overline{v_{i_j} v_{i_{j+1}}})$$

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.

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 $\mathcal{C}_i$.

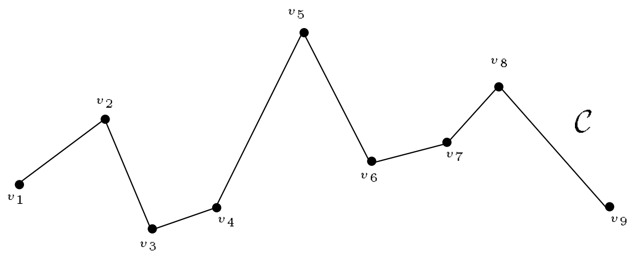


Figure 1: A single chain $\mathcal{C} = \{v_1, \ldots, v_9\}$ of size 9.

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 quantity $\sum_i$ size($\mathcal{C}'_i$) 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 Figure 2 for one valid simplification of the chain shown in Figure 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.

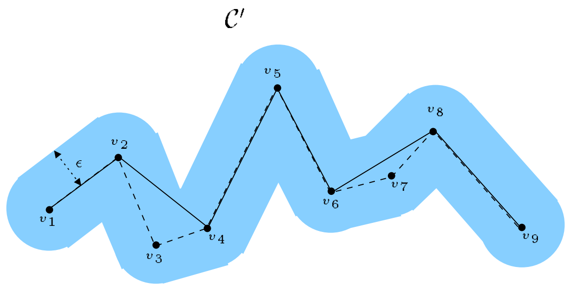


Figure 2: A simplified chain $\mathcal{C}' = \{v_1, v_2, v_4, v_5, v_6, v_8, v_9\}$ (solid line) of input chain $\mathcal{C}$ (dotted line), with $size(\mathcal{C}') = 7$, and $d(\mathcal{C}', \mathcal{C}) \leq \epsilon$.

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 visual fidelity (expressed by the distance constraint). It is important to note that there are two separate components of similarity, namely 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.

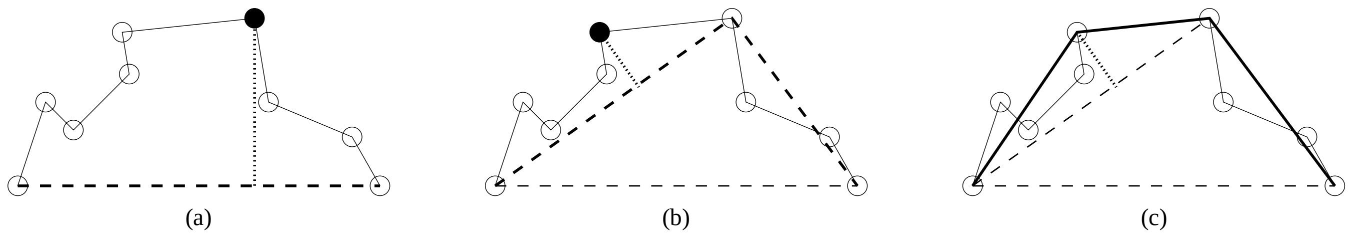


Figure 3: **An illustration of the Douglas-Peucker algorithm.**

3.1 Cartography

The problem of map simplification[1] is a central concern of the cartographic community. Traditionally, the focus has been on subjective measures of simplification such as aesthetic considerations and the retention of (subjectively decided) important features. However, as far back as the early 70s, there was a growing interest in automated methods, which necessarily required objective measures of goodness. One of the first and most effective methods for map simplification was the algorithm proposed by Douglas and Peucker [8] in 1973. This algorithm was very efficient, and it was observed by researchers [32, 26] that the simplifications produced by this method tended to conform to the subjective solutions produced by domain experts. We employ some ideas from this algorithm in our system and present a brief overview of it here.

The algorithm works by successive refinement of an initial solution (see Fig. 3) using a greedy approach. Let ϵ be the tolerance for a given chain. The initial solution (see Fig. 3(a)) is obtained by forming the line segment connecting the endpoints of the chain. If all vertices of the chain are within distance ϵ of this solution (the dashed line), the algorithm terminates. Otherwise, it finds the vertex that is furthest from the line segment (the black vertex), and creates a refinement consisting of the two line segments connecting this point to the endpoints (Fig. 3(b)). This process is repeated recursively on each of the segments. It is not difficult to see that this algorithm always terminates. The final solution (in bold) is shown in Fig. 3(c). A detailed description is presented in Algorithm 1. A straightforward implementation of the algorithm runs in time $O(k^2)$ for a chain of size k; Hershberger and Snoeyink [17] 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 [10], is to run the Douglas-Peucker algorithm first, and use a clever local improvement technique to eliminate intersections.

Algorithm 1 `Douglas_Peucker`(Input Chain $\mathcal{C}$, Tolerance Parameter ϵ)
Input: Polygonal chain $\mathcal{C} = \{v_1, v_2, \ldots, v_m\}$ and tolerance parameter ϵ
Output: Polygonal chain $\mathcal{C}' = \{v_1 = v_{j_1}, v_{j_2}, \ldots, v_{j_{m'}} = v_m\}$, with $d(\mathcal{C}', \mathcal{C}) \leq \epsilon$

```
$Q \leftarrow \{(v_1, v_m)\}$
$\mathcal{C}' \leftarrow \emptyset$
while $Q \neq \emptyset$ do
  /* pick any shortcut segment */
  Remove $(v_i, v_j)$ from $Q$
  dmax ← 0
  vmax ← 0

  /* Find furthest vertex from shortcut segment $\overline{v_i v_j}$ */
  for $k = (i+1)$ to $(j-1)$ do
    dist ← $d(v_k, \overline{v_i v_j})$
    if dist > dmax then
      dmax ← dist
      vmax ← k
    endif

  if dmax > $\epsilon$ then
    /* $\overline{v_i v_j}$ is not a good approximation */
    Insert $(v_i, v_{\text{vmax}})$ and $(v_{\text{vmax}}, v_j)$ into $Q$
  else
    /* $\overline{v_i v_j}$ is an edge in the solution */
    $\mathcal{C}' = \mathcal{C}' \bigcup \{v_i, v_j\}$
endwhile
Return $\mathcal{C}'$
```

3.2 Computational Geometry

Map simplification (and the related problem of chain simplification where each map consists of only one chain) has been studied extensively by computational geometers [18, 7, 4, 15, 2]; recent work [9] has shown that map simplification is NP-hard. Moreover, Estkowski [10] has 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 [4] 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 map simplification algorithms. To the best of our knowledge, there exists only one implementation (of the algorithm proposed by Estkowski [10]). 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.

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 [15]; 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 mesh simplification [31, 29, 20, 19, 5, 11, 24, 28] has been studied extensively.

Several of the mesh simplification algorithms rely on succes-

[1]Also referred to as *map generalization* in the cartography literature.

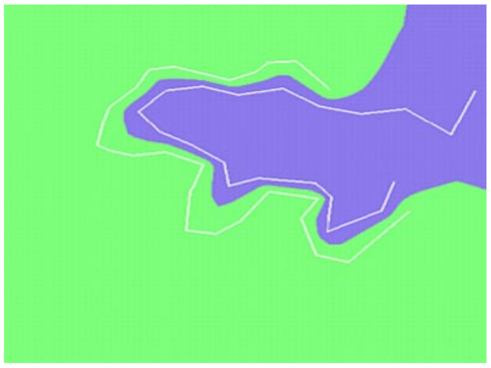


Figure 4: Voronoi regions of two chains that partition the plane.

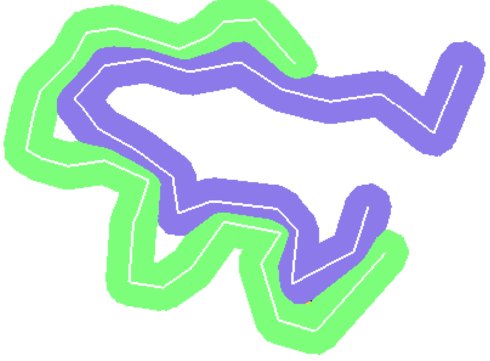


Figure 5: ε-Voronoi regions

sive vertex removal and local triangulation to simplify the meshes. Mesh decimation [31] and vertex clustering [29] are examples of such techniques. Simplification using local edge collapses and edge swaps were used by Hoppe et. al. and Garland et. al. [20, 19, 11, 28]. Cohen at. al [5] use the notion of *simplification envelopes* to generate genus-preserving simplifications. More recently, Lindstrom et. al. [24] use image similarity to perform mesh simplification.

However, mesh simplification algorithms are not directly applicable to our problem. This is because mesh simplification algorithms work on either manifolds or simplicial complexes, which are topologically different from maps (which are two dimensional subdivisions).

There have been some studies of exploiting the power of graphics hardware to implement geometric algorithms. Rossignac et. al. [30] and Goldfeather et. al. [12] use hardware to perform real-time constructive solid geometry. Greene et. al. [14, 13] have used the "Z query" feature of some specialized graphics hardware to implement their hierarchical Z-buffering algorithm. Finally, 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.

4. OVERVIEW OF OUR METHOD

In this section, we discuss briefly 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; we will call such segments *valid* shortcut segments.

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. However, for large maps this quadratic size can be prohibitive, and we use a heuristic to choose a small set of "good" shortcut segments. Once we have an initial set of candidate shortcut segments, we need to cull out the invalid segments from them.

For a shortcut segment to be valid, it must satisfy both geometric and topological conditions. Any shortcut segment that might be present in a valid simplification of a chain $\mathcal{C}$ must be within distance ε of $\mathcal{C}$. 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 semgents 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 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*. Figure 4 shows the compliant regions of a map.

It is now easy to see that all shortcut segments that are both proximate and compliant are valid.

Now consider the region consisting of all points within distance ε of $\mathcal{C}$ i.e the region $\mathcal{C} \oplus B_\varepsilon$ where B_ε is the ball of radius ε with respect to the underlying distance function, and $\oplus$ is the standard Minkowski sum. Figure 6 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 Figure 6. 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

$$d_\varepsilon(p,q) = \begin{cases} d(p,q) & d(p,q) \leq \varepsilon \\ \infty & \text{otherwise} \end{cases} \qquad (1)$$

An example of ε-Voronoi diagram of two chains is shown in Figure 5. The advantage of using ε-Voronoi regions is that they are subsets of the corresponding Voronoi region. Further, the nonproximate shortcut segments that do not lie within a Minkowski region $\mathcal{C} \oplus B_\epsilon$ will also not lie in a ε-Voronoi region (due to the definition

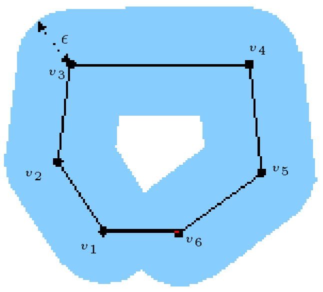


Figure 6: An example of a non-proximate segment $\overline{v_1 v_6}$ (bold) completely inside the region defined by the Minkowski sum.

Algorithm 2 `MapSimplification`(Input Map $\mathcal{M}$, Tolerance Parameter Array $\mathcal{E}$)
Input: Map $\mathcal{M} = \{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$, Tolerance parameters $\mathcal{E} = \{\varepsilon_1, \ldots, \varepsilon_n\}$
Output: Simplified map $\mathcal{M}' = \{\mathcal{C}'_1, \ldots, \mathcal{C}'_n\}$, with $d(\mathcal{C}'_i, \mathcal{C}_i) \leq \varepsilon_i$

```
/* Initialize set of shortcut segments $\mathcal{S}_i$ for each chain*/
for Chain $\mathcal{C}_i \in \mathcal{M}$ do
    $\mathcal{S}_i$ = Find_Base_Segments ( $\mathcal{C}_i$ )

Compute the $\varepsilon$-Voronoi diagram of $\mathcal{M}$.

/* Remove set of non-compliant segments from the map */
Remove_NonCompliant_Segments( $\mathcal{M}$ ).

for Chain $\mathcal{C}_i \in \mathcal{M}$ do
    /* Remove non-proximate segments from each chain */
    Remove_NonProximate_Segments( $\mathcal{C}_i$, $\mathcal{S}_i$ ).

for Chain $\mathcal{C}_i$ in $\mathcal{M}$ do
    /* Find shortest path by shortcut segments $\mathcal{S}_i$ of $\mathcal{C}_i$. */
    $C'_i$ = Find_Shortest_Path( $\mathcal{C}_i$, $\mathcal{S}_i$ ).
```

of the distance measure for ε-Voronoi region). Thus, these segments can now be rejected, without having to do proximity testing on them.

Given a set of valid shortcut segments for a chain, determining a valid simplification is straightforward. Observe that the minimum sized simplification (with respect to the set of base shortcut segments) is precisely a shortest path from the start vertex to the end vertex of the chain using the valid shortcut segments. Notice that this step introduces the possibility of self-intersections. Recently, Mantler and Snoeyink [25] proposed a way of partitioning chains into so-called *safe sets* so that *any* simplification method is guaranteed not to create self-intersections. It is possible that this technique can be incorporated into our system.

Algorithm 2 summarizes the above discussion. We now discuss each of these functions in more detail.

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. 7. We will use this example to illustrate the working of our algorithm at each step.

5.1 Preprocessing Step: Computing the set of candidate shortcut segments

The algorithm is provided with a set of candidate shortcut segments $\mathcal{S}_i$ for each chain $\mathcal{C}_i$. The simplest way to generate this set for each chain is to compute all $\binom{m}{2}$ pairwise segments for a chain of size m. While this provides a complete set of segments for computing our simplification, its quadratic size is prohibitively expensive.

However, in practice, if we can choose a "good" set of base shortcut segments, whose size is smaller, we gain significant improvements in speed. We use this idea to select a linear-sized set of base shortcut segments.

As mentioned before, the Douglas-Peucker algorithm is well-

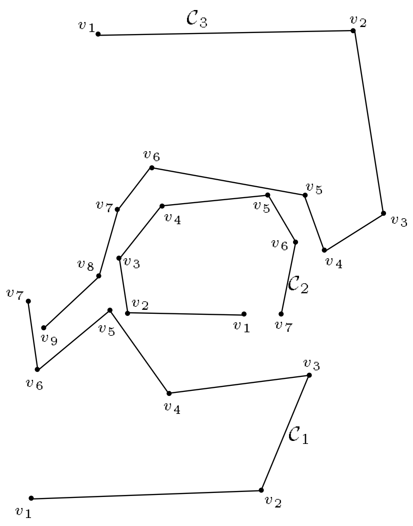


Figure 7: Input chains (non-intersecting) to demonstrate our algorithm.

known for producing visually pleasing simplifications [32, 26]. Now consider the Douglas-Peucker algorithm described in Algorithm 1. Notice that once the algorithm finds a segment which is proximate, it stops the recursion, and adds that segment to the simplified chain. However, we modify Algorithm 1 in the following way: instead of retaining only the proximate segments, we retain *all* shortcut segments obtained during a simulation of the Douglas-Peucker algorithm. This idea was first proposed by Cromley [6]. We present the details of this method in Algorithm 3.

The advantages of using the modified Douglas-Peucker algorithm are:

- **Linear sized set of shortcut segments.** It is not difficult to show that the set of shortcut segments added to our bag of base segments is linear-sized. Therefore, the size decreases from $\binom{m}{2}$ to $2m$-1.

- **Computation of $\Delta(\overline{v_i v_j})$.** We can obtain $\Delta(\overline{v_i v_j})$ as a side product of Douglas Peucker algorithm. Note that for each segment $\overline{v_i v_j}$ retained, the distance of the furthest vertex of the chain from this segment, $\Delta(\overline{v_i v_j})$, is the value of ε above which this segment becomes proximate. Thus, this first step eliminates the need to explicitly determine (for a given ε) which shortcut segments are proximate.

Algorithm 3 is run for each chain as a preprocessing step, generating the base set of shortcut segments $\mathcal{S}_i$ for each chain $\mathcal{C}_i$.

The storage required by this preprocessing step is bounded by $O(|\mathcal{M}|)$, where $|\mathcal{M}|$ is the complexity of the input map, and thus the total memory requirement is linear in the input size. Figure 8 shows the candidate set determined by the algorithm, together with the input chains.

5.2 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 (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]. Their work is applicable to sets of points and polylines. For each chain

Algorithm 3 `Find_Base_Segments`(Input Chain $\mathcal{C}_i$)
Input: A Chain $\mathcal{C}_i == \{v_1, \ldots, v_{m_i}\}$
Output: The set of base shortcut segments $\mathcal{S}_i$

```
$Q \leftarrow \{(v_1, v_{m_i})\}$
$\mathcal{S}_i \leftarrow \emptyset$
while $Q \neq \emptyset$ do
    /* pick any shortcut segment */
    Remove $(v_i, v_j)$ from $Q$
    dmax $\leftarrow 0$
    vmax $\leftarrow -1$

    /* Find furthest vertex from shortcut segment $\overline{v_i v_j}$ */
    for $k = (i+1)$ to $(j-1)$ do
        dist $\leftarrow d(v_k, \overline{v_i v_j})$
        if dist $\geq$ dmax then
            dmax $\leftarrow$ dist
            vmax $\leftarrow$ k
        endif

    if vmax $\neq -1$ then
        /* Associate dmax with segment $\{v_i, v_j\}$ */
        $\Delta(\overline{v_i v_j}) = dmax$
        Insert $(v_i, v$vmax$)$ and $(v$vmax$, v_j)$ into $Q$
        /* Add $\overline{v_i v_j}$ as a potential edge for some $\varepsilon$ */
        $\mathcal{S}_i = \mathcal{S}_i \bigcup \{v_i, v_j\}$
    endif
endwhile
Return $\mathcal{S}_i$
```

$\mathcal{C}_i$, they compute a set of 3-dimensional primitives based on the vertices and line segments of that chain (the reader is referred to their paper for details), and render all the primitives of a chain $\mathcal{C}_i$ with a unique color, say c_i. Once all the primitives of all the chains are rendered, they use the Z-buffer to compute the lower envelope of these primitives efficiently. The color buffer then gives the Voronoi diagram for the chains, where the region in the color buffer with color c_i is the Voronoi region for the chain $\mathcal{C}_i$. 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 $\mathcal{C}_i$ [2]. See Figure 9 for an example of the stencil buffer state with the corresponding Voronoi diagram in the color buffer after this phase is finished, and Figure 10 for the ε-Voronoi diagram of our three chains.

5.3 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.

[2] 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 [16] for most data sets.

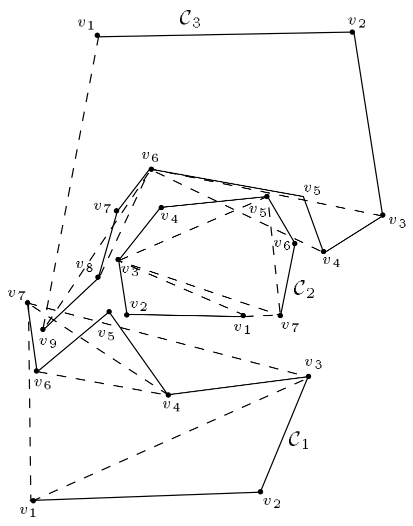
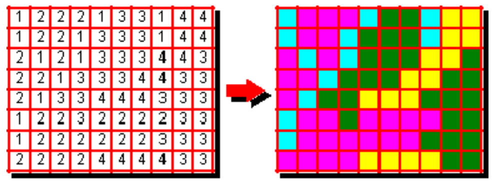


Figure 8: Preprocessing: Input chains (solid) together with the initial set $\mathcal{S}$ of candidate segments (dashed).



1	2	2	2	1	3	3	1	4	4
1	2	2	1	3	3	3	1	4	4
2	1	2	1	3	3	3	4	4	3
2	2	1	3	3	3	4	4	3	3
2	1	3	3	4	4	4	3	3	3
1	2	2	3	2	2	2	2	3	3
1	2	2	2	2	2	2	3	3	3
2	2	2	2	4	4	4	4	3	3

Figure 9: Stencil region with corresponding color buffer.

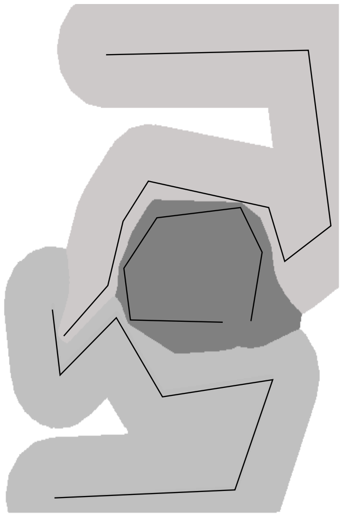


Figure 10: Step 1: ϵ-Voronoi diagram of the chain segments.

Algorithm 4 `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 $\mathcal{S} = \{\mathcal{S}_1, \ldots, \mathcal{S}_m\}$.

```
repeat
  for Chain $\mathcal{C}_i \in \mathcal{M}$ do
    glStencilFunc( GL_FAIL, GL_FAIL, GL_PASS )
    glStencilTest( GL_NEQUAL, i )
    for Shortcut segment $\{v_j, v_k\} \in \mathcal{S}_i$ do
      Set( Color of $\{v_j, v_k\}$ )
      Draw($\{v_j, v_k\}$)
  ColorBuffer = ReadColorBuffer( )
  for Unique color $c$ in ColorBuffer do
    Remove segment whose color is $c$ from $\mathcal{S}$.
until $\mathcal{S}$ does not change
```

Note that if the number of shortcut segments, say $|\mathcal{S}|$, is greater than $2^{32} - 1$, the algorithm given in Algorithm 4 would have to be repeated $\lceil \frac{|\mathcal{S}|}{2^{32}-1} \rceil$ times.[3]

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.

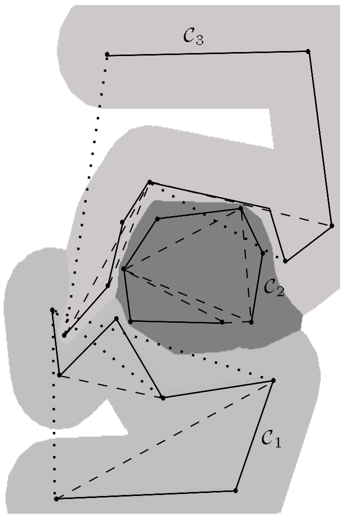


Figure 11: Step 2: Set of compliant shortcut segments (dashed) and non-compliant shortcut segments (bold dotted)

The pseudocode for the algorithm is shown in Algorithm 4. Figure 11 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).

[3] we assume an 8-bit per channel 4-channel color buffer (RGBA).

5.4 Step 3: Removing remaining nonproximate shortcut segments

As observed in Section 4, there are cases (e.g. Figure 6) where the previous phase cannot detect and remove all the nonproximate shortcut segments. Suppose the set of shortcut segments output by Step 2 is the set $\mathcal{S}$. We mentioned in the preprocessing step that for each shortcut segment $\overline{v_i v_j}$ we maintain its maximum distance ($\Delta(\overline{v_i v_j})$) to the original chain. Given all the compliant segments, we compare this value with the corresponding tolerance parameter for the chain and determine if it is proximate or not. Figure 12 shows the proximate segments that are retained after this step.

5.5 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. Notice however that the edges of this graph are not arbitrary; they form a subset of the edges encountered in the Douglas-Peucker algorithm. Thus, instead of computing a breadth-first traversal of the graph, we can use a greedy algorithm that from the current node v (the first node initially) moves to the node that is the *furthest* number of links away from v in the chain (moving from the beginning to the end).

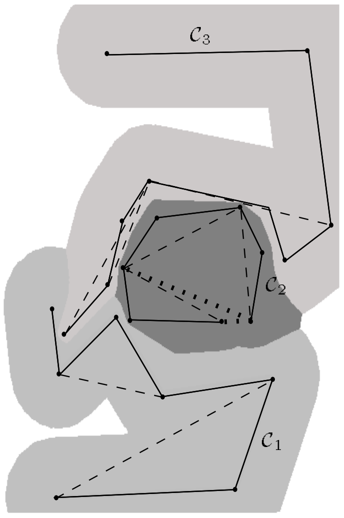


Figure 12: Step 3: Set of compliant proximate shortcut segments (dashed) and compliant nonproximate segments (bold dotted).

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 $\mathcal{C}_i$. Since the path is a shortest path, the simplification is optimal for the set of valid shortcut segments.

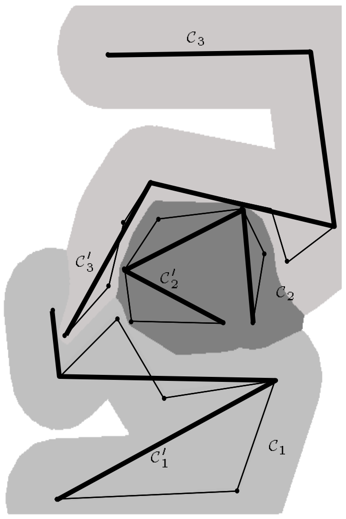


Figure 13: Step 4: Simplified chains (bold) and Input chains.

Figure 13 shows the final simplification produced by the algorithm.

5.6 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*.

Our algorithm can be modified to respect certain types of point constraints. Consider a point constraint $(p, \mathcal{C})$, where p is a point and $\mathcal{C}$ is the constrained chain. For each such constraint, we treat p as a chain (of length 0) having a tolerance parameter equal to that of $\mathcal{C}$[4]. This ensures that the chain cannot *flip* sides. If there are multiple chains constrained by a point, then we choose the tolerance to be the maximum of the chain tolerances.

Suppose there are two chains $\mathcal{C}_1, \mathcal{C}_2$ near a point p, but only the constraint $(p, \mathcal{C})$ is present. This might force $\mathcal{C}_2$ to be simplified to a lesser degree than possible. In general though, if a point is constrained with respect to all chains surrounding it, the above approach will not over-constrain the problem.

6. MAINTAINING INTERACTIVITY

The map simplification algorithm is one module of our visualization system. The run-time system consists of three main modules: the *display module* that renders the simplified chains and provides user interactivity, the *selection module* that takes the current viewing parameters and generates a set of chains to be simplified, and the *simplification module* described in the previous section, which simplifies a given set of chains and returns the output to the display routine. The system iterates through each of these modules at each frame; Figure 14 illustrates the system components and their interaction.

The system also has a preprocessing module, which is invoked once when the system is initialized. We start by describing this module.

[4]We only need the tolerance for the point to be at least $\varepsilon - d(p, \mathcal{C})$, which is always at most ε.

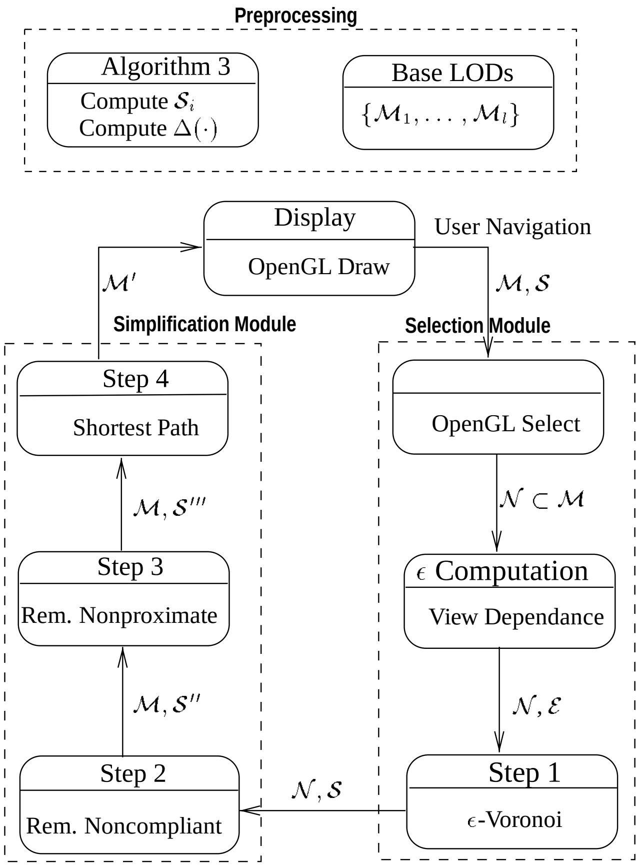


Figure 14: System components of our Map viewer

6.1 Preprocessing Module

Firstly, we compute an initial set of shortcut segments for each chain in the map using the procedure described in Section 5.1. Along with this information, we also maintain the bounding box of each input chain. Next, for a fixed set of tolerance values $\{\varepsilon_1 < \varepsilon_2 < \ldots < \varepsilon_l\}$, we invoke the simplification module on the map $\mathcal{M}$, obtaining a set of simplifications $\mathcal{M}_j$. For a chain $\mathcal{C}_i$, the simplified chain $\mathcal{C}'_{i,j} \in \mathcal{M}_j$ represents the base level-of-detail (LOD) for $\mathcal{C}_i$ for values of ε between ε_j and ε_{j+1} (note that different chains can have different ε values). We use these base LODs as the input into our run-time simplification system (which produces continuous simplifications).

6.2 The Selection Module

The selection module is the first stage of the run-time system. It takes as input the chain bounding boxes generated by the preprocessing phase and information about the current viewing parameters.

For large maps, typically only a small portion of the map is visible at any given time. Therefore, significant performance gains can be achieved by eliminating portions of the map outside the viewport. Using the OpenGL *GL_SELECT* feature, we eliminate all chains that are not currently visible.

Chains that are closer to the viewer should appear in more detail than chains that are further away. The selection module creates this effect by computing a view-dependent tolerance parameter for each chain, giving chains that are closer a smaller tolerance. Note that

ϵ	USA	TEXAS	EURASIA
0.0	261,460	151,983	222,969
0.0002	9,718	14,232	46,792
0.0008	3,287	5,154	20,094
0.0040	1,233	2,173	7,867

Table 2: Sizes of simplified maps as tolerance varies

the mechanism for generating tolerances is modular; other effects can be created as well, such as focusing on user-specified chains.

Using these chain-specific tolerance parameters, the selection module can now pick the appropriate base LOD for each chain. Specifically, if the tolerance for chain $\mathcal{C}_i$ is ε, then it picks the LOD $\mathcal{C}'_{i,j}$ where $\varepsilon_j < \varepsilon \leq \varepsilon_{j+1}$. Note that for different chains we may have different LODs; this results from having a view where some chains are closer than others.

Finally, for each LOD chosen, the selection module scans its set of shortcut segments (generated during the preprocessing step) and eliminates all those segments that are not proximate with respect to the tolerance for that chain. Note that our technique for generating shortcut segments ensures that for each segment, we know the value of ε above which it is proximate.

The Display Module The chains output by the selection module are sent to the simplification module and the simplified chains are rendered by the display module, where all user interaction takes place.

7. PERFORMANCE ANALYSIS

In this section we report the results of experiments testing the performance and quality of our system. Our system was built using C++ over the OpenGL library on an SGI Onyx with an R10000 processor with 2 GB of main memory.

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. The second map (called TEXAS) is a detailed map of the state of Texas, containing both state and county lines. This map has 151,983 vertices and 817 chains. The third map (called EURASIA) is a detailed political map of Europe and Asia, containing both state boundaries as well as country boundaries. This map has 222,969 vertices and 2,674 chains. Due to lack of space, we do not display these maps here. The reader may go to [1] to see them as rendered.

Table 2 shows the level of simplification (in size of the simplified map) achieved for different values of the tolerance parameter ε, and Figures 15-20 at [1] show the simplified maps. Observe that visual fidelity is preserved, even when the number of vertices rendered are much smaller than that of the original map.

In the next set of experiments, we evaluate the performance of our method. Firstly, we report the frame rates achieved by our system while navigating through the maps. We compare this to the interactivity achieved when no simplification is performed. For this experiment, we use three maps: the maps USA and TEXAS described above, and a detailed map of the United States (called USA-FULL) that contains both state and county boundaries. This map has $1,685,300$ vertices and 9,729 chains.

Our results are reported in Table 3. Observe that for all the maps, the frame rate achieved by our system is superior to that achieved by a simple viewer that merely renders the unsimplified map, inspite of the extensive simplification computations we perform at each frame. Observe that this discrepancy increases with the size of the map; for the map USA-FULL, the frame rate of the simple viewer is 0.91, whereas ours is 8.

	TEXAS	USA	USA-FULL
Our algorithm	22-30	14-20	8
Simple viewer	11	6	0.9

Table 3: Frame rates for various maps (as frames/s)

Finally, we present data on the relative amount of time spent on each step in the display cycle. Referring back to Figure 14, there are five main steps in the rendering pipeline: OpenGL SELECT, view-dependent edge selection (proximity checking), Voronoi diagram construction, compliance checking, and shortest path computation. In Table 1 we present the relative fraction of time spent in each of these steps. The timings were generated (for each data set) by simulating a series of navigation paths in our viewer. The specific nature of the motion tends to have little effect on the percentage time spent. We note here that preprocessing times are negligible for these maps; typical running time for the preprocessing phase is 0.5-1 second for the maps USA, TEXAS, and EURASIA, and 3 seconds for USA-FULL.

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 major challenge is scaling our system to handle maps that have billions of vertices (street-level maps of the United States for example). Currently, our system requires the entire map to be processed in-core, which is not feasible for such maps. Another challenging open problem is to integrate our system with data visualization systems (like Swift3D [22]) so that at varying levels of detail, the user may issue queries to the system for various types of data.

The paradigm of simulating general purpose computations on the graphics pipeline is a recent trend in graphics and visualization [23, 13, 21, 27]. 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. ACKNOWLEDGEMENTS

We would like to thank Dinesh Manocha and Tim Culver for providing us with the code for computing Voronoi diagrams. We would also like to thank Jack Snoeyink, Andrea Mantler and Claudio Silva for helpful comments.

10. REFERENCES

[1] Maps of the United States, Texas and Eurasia. `http://www.cs.duke.edu/~nabil/Simp`.

[2] P. Agarwal, S. Har-Peled, and N. Mustafa. Efficient approximation algorithms for simplifying polygonal chains. In *10th Fall Workshop on Computational Geometry*, 2000.

[3] United States Census Bureau. TIGER: Topologically integrated geographic encoding and referencing system. `http://www.census.gov/geo/www/tiger/`.

	TEXAS	USA	USA-FULL	EURASIA
OpenGL SELECT	7	6	6	5
Proximity checking	9	10	17	17
Voronoi diagram computation	22	18	30	19
Compliance checking	54	60	37	48
Shortest path computation	8	6	9	10

Table 1: Relative times spent in each step (in %)

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

[5] Jonathan Cohen, Amitabh Varshney, Dinesh Manocha, Greg Turk, Hans Weber, Pankaj Agarwal, Frederick P. Brooks, Jr., and William Wright. Simplification envelopes. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 119–128. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[6] R. G. Cromley. Hierarchical methods of line simplification. *Cartography and Geographic Information Systems*, 18(2):125–131, 1991.

[7] M. de Berg, M. van Kreveld, and S. Schirra. A new approach to subdivision simplification. In *ACMS/ASPRS Annual Convention and Exposition*, volume 4, pages 79–88, 1995.

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

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

[10] R. Estkowski. Subdivision simplification : Hardness of approximation and a heuristic. Manuscript, 1999.

[11] M. Garland and P. Heckbert. Surface simplification using quadric error bounds. *Proc. of ACM SIGGRAPH*, pages 209–216, 1997.

[12] Jack Goldfeather, Steven Molnar, Greg Turk, and Henry Fuchs. Near real-time CSG rendering using tree normalization and geometric pruning. *IEEE Computer Graphics and Applications*, 9(3):20–28, May 1989.

[13] N. Greene, M. Kass, and G. Miller. Hierarchical Z-buffer visibility. In *Proc. 20th ACM SIGGRAPH*, 1993.

[14] Ned Greene and M. Kass. Hierarchical Z-buffer visibility. In *Computer Graphics Proceedings, Annual Conference Series, 1993*, pages 231–240, 1993.

[15] Leonidas J. Guibas, J. E. Hershberger, Joseph S. B. Mitchell, and J. S. Snoeyink. Approximating polygons and subdivisions with minimum link paths. *Internat. J. Comput. Geom. Appl.*, 3(4):383–415, 1993.

[16] E. Halperin, R. Nathaniel, and U. Zwick. Coloring k-colorable graphs using smaller palettes. In *Proc. 12th ACM-SIAM Symposium on Discrete Algorithms*, 2001.

[17] J. Hershberger and J. Snoeyink. Speeding up the Douglas-Peucker line simplification algorithm. In *Proc. 5th Internat. Sympos. Spatial Data Handling*, pages 134–143, 1992.

[18] J. Hershberger and J. Snoeyink. Computing minimum length paths of a given homotopy class. *Comput. Geom. Theory Appl.*, 4:63–98, 1994.

[19] Hugues Hoppe. Progressive meshes. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 99–108. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[20] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh optimization. In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 19–26, August 1993.

[21] K. E. Hoff III, T. Culver, J. Keyser, M. Lin, and D. Manocha. Fast computation of generalized Voronoi diagrams using graphics hardware. In *Proc. ACM SIGGRAPH*, 1999.

[22] E. K. Koutsofios, S. C. North, R. Truscott, and D. Keim. Visualizing large-scale telecommunication networks and services. In *IEEE Visualization*, 1999.

[23] S. Krishnan, C. T. Silva, and B. Wei. Hardware-assisted visibility-ordering algorithm with applications to volume rendering. In *Eurographics/IEEE Symposium on Visualization*, 2001.

[24] P. Lindstrom and G. Turk. Image-driven simplification. *ACM Transactions on Graphics*, 19(3), 2000.

[25] A. Mantler and J. Snoeyink. Safe sets for line simplification. In *10th Annual Fall workshop on Computational Geometry*, 2000.

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

[27] M. S. Peercy, M. Olano, J. Airey, and P. J. Ungar. Interactive multi-pass programmable shading. In *Proc. 27th ACM SIGGRAPH*, 2000.

[28] J. Popovic and H. Hoppe. Progressive simplicial complexes. In *Proc. ACM SIGGRAPH*, pages 217–224, 1997.

[29] J. Rossignac and P. Borrel. Multi-resolution 3D approximation for rendering complex scenes. In *Second Conference on Geometric Modelling in Computer Graphics*, pages 453–465, June 1993. Genova, Italy.

[30] J. R. Rossignac and A. A. G. Requicha. Depth-buffering display techniques for constructive solid geometry. *IEEE Computer Graphics and Applications*, 6(9):29–39, 1986.

[31] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. In Edwin E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 65–70, July 1992.

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