Lecture XI
Geometric and Geographic Simplification

This lecture introduces two related and very important topics. In visualization, the issue is how to simplify geometric models. In GIS, the issue is how to simplify map data. They are clearly related, but the relationship is not simple to describe.

We have already seen in our homework that displaying a map in all its details is too much for a dynamic zooming display. What is needed is the ability to adjust the amount of details to be displayed at different zoom levels.

§1. Introduction

In visualization, we are given a geometric model M to be displayed. This model can be a map (2-D) or it can be a 3-D model. We assume that there is some measure of **size** of the model, denoted $|M|$. The main issues of visualizing M arises becase $|M|$ can be very large.

The next idea is that M might be **approximated** by other models M' where $|M'| < |M|$. We assume there is some measure of their difference, $d(M, M') \geq 0$. Let $\delta \geq 0$ and $\sigma \geq 1$. We will call M' a **(δ, σ)-simplification** of M if $d(M, M') \leq \delta$ and $|M| \leq \sigma |M'|$. We will also say M' is a **δ-simplification** of M.

There are two ways to go defining M' for a given M: we could specify δ and then seek the smallest σ such that M' is a **(δ, σ)-simplification** of M. Or we could specify σ and seek the smallest δ such that this relation holds. Of course, finding the "smallest" is going to be a hard optimization problem. So we may be content with some good heuristics.

Suppose $\delta > 0$ is fixed, and we have an algorithm that for each M_i, create a new M_{i+1} that is a δ-simplification. This will allow us to construct a **hierarchy** of models,

$$M_0, M_1, \ldots, M_k.$$

We can use this hierarchy for visualization: depending on the zoom level or desired level of detail, we use the appropriate model M_i.

But how do we do simplification? We begin with two basic paradigms, called **decimation** and **clustering**, respectively. Let us illustrate them with the simplest kind of geometric model, where M is a polygonal path. So $M = (v_0, v_1, \ldots, v_n)$ where v_i are points in the plane.

(1) Decimation: we can define $M' = (v_0, v_2, v_4, \ldots, v_{2\lfloor n/2 \rfloor})$. Thus, we simply drop every other point in the path.

(2) Clustering: let us fix a grid G in the plane. View the grid as a discrete set of points, $G \subseteq \mathbb{R}^2$. we can define $C(v_i)$ to be the closest grid point to v_i (breaking ties in some systematic way). Then we can replace M by $(C(v_0), C(v_1), \ldots, C(v_n))$, and further dropping any $C(v_i)$ in case $C(v_i) = C(v_{i+1})$. The result is M'.

The difference in these two points of views is that decimation guarantees reduction in the size $|M| = n$ while clustering guarantees a bound in the distance (or distortion) $d(M, M')$.

§2. The Douglas Peucker Algorithm

The Douglas Peucker (DP) Algorithm [2] is a method to simplify a polygonal path, originally designed for simplifying map data. Psychological studies have suggested that it produces very good perceptual representations of the original lines. It is a simple and natural algorithm, and so have been re-discovered several times in other literature. See [3] for this and related literature.

The DP Algorithm solves the following problem: given a polygonal path $p = (v_0, v_1, \ldots, v_n)$, $n \geq 2$, and an error bound $\varepsilon > 0$, compute a subsequence $p' = (v'_0, v'_1, \ldots, v'_m)$ such that $d(p, p') \leq \varepsilon$, and $v'_0 = v_0, v'_m = v_n$.

Note that "subsequence" means that p' is obtained from p be omitting 0 or more vertices (so $m \leq n$). We define $d(p, p')$ as follows: if $|p'| = 2$, then $d(p, p')$ is just the maximum distance of any point v_i from the line $\overline{v_0 v_n}$. If $|p'| > 2$, then there is ome $0 < k < m$ such that $p' = p_1; p_2$ (where ; denotes path concatenation) where $p_1 = (v'_0, \ldots, v'_k)$ and $p_2 = (v'_k, \ldots, v'_m)$ and recursively,

$$d(p, p') = \max\{d(p, p_1), d(p, p_2)\}.$$

To compute this distance measure, it is useful to recall that if a, b, c are three points, then the distance of c from the line $\overline{ab}$ is

$$d(a, b, c) = \frac{|\Delta(a, b, c)|}{\|a - b\|} \quad \text{where}$$

$$\Delta(a, b, c) \quad = \quad \det \begin{bmatrix} a.x & a.y & 1 \\ b.x & b.y & 1 \\ c.x & c.y & 1 \end{bmatrix}$$

Note that $\|a\| = \sqrt{a.x^2 + a.y^2}$ is the Euclidean norm.

A simple implementation of the DP Algorithm is as follows

DP ALGORITHM
INPUT: an array V of n vertices, and indices $1 \leq i < j \leq n$
and $\varepsilon \geq 0$
OUTPUT: A subsequence U of $V[i, j]$ such that $d(U, V[i, j]) \leq \varepsilon$
METHOD:
0. If $j - i = 1$ return $(V[i], V[j])$.
1. Find the vertex v_f that is farthest from $\overline{V_i V_j}$.
   Let *dist* be this distance.
2. If $dist \leq \varepsilon$
   then Return $U = (V_i, V_j)$
3. Recursively,
   Return $DP(V, i, f, \varepsilon); DP(V, f, j, \varepsilon)$

We assume that computing v_f takes $O(j - i + 1)$ time. It is not hard to see that this algorithm takes $\Theta(n^2)$ in the worst case. Hershberger and Snoeyink [3] shows how to improve this to $O(n \log n)$. They further show that the theoretical bound of $O(n \log^* n)$ is possible [4].

The DP Algorithm outputs a path p' that may not preserve simplicity of the input path p. There are more sophisticated algorithms that can preserve simplicity and achieve other additional properties. For instance, we can compute the simplified p' of minimum length (subject to $d(p, p') \leq \varepsilon$). Conversely, given a target $m < n$, there are algorithms to give the best subsequence p' of length m such that $d(p, p')$ is minimized. Moreover, alternative measures of distortion that are different from $d(p, p')$ can be investigated.

§3. The Douglas Peucker Hierarchy

The DP line simplification algorithm can be used to to generate an "integrated hierarchy" that can be used in visualization. What we mean by integrated hierarchy is this: if $(P_0, P_1, \ldots, P_h)$ is a seqence of simplifications of P_0 (so P_{i+1} is a simplification of P_i, then we call this a "simplification hierarchy". But this is not integrated yet. Here, we are assuming that simplication is by decimation.

Let $P_0 = (v_1, \ldots, v_n)$ be a path with n vertices. A **DP hierarchy** for a path P_0 is a binary tree H with $2n-2$ nodes. Each node u stores (1) a subpath of P_0 denoted $span(u)$ and (2) a **split vertex** $v(u)$. If u is a leaf, then $v(u)$ is undefined and $span(u)$ is a pair (v_i, v_{i+1}) $(i = 1, \ldots, n-1)$ of consecutive vertices of P_0. If u is an internal node with u_L, u_R as left and right children, then $span(u) = span(u_L); span(u_R)$, and $v(u)$ is the endpoint of $span(u_L)$ and the startpoint of $span(u_R)$. Note that $P; Q$ denotes concatenation of paths P and Q and this is defined provided the endpoint of P equals the startpoint of Q and the resulting path is just obtained by concatenating the correspond two sequences, after removing the startpoint of Q (to avoid the obvious duplication). It is clear that the DP algorithm produces such a hierarchy.

An **antipath** is a sequence $(u_1, \ldots, u_m)$ of nodes in T such that every path from the root of T to a leaf must intersect this sequence. We say that H contains the hierarchy $(P_0, \ldots, P_h)$ if each P_i can be extracted as an antipath of H.

We can construct a DP hierarchy by calling the DP algorithm with $\varepsilon = 0$. The modifications to DP algorithm are
(1) We assume he DP Algorithm now returns the root of a DP Hierarchy on the input (sub)path. We construct the DP hierarchy as we go. Each time we find a split point v_f, we create a new node u with $v(u) = v_f$, and make its results of the two recursive calls the children of u. (2) When $j - i = 1$ we simply create a leaf node with no split nodes and with span (v_i, v_j).

How do we use a DP hierarchy H? Given a $\varepsilon > 0$, we can recursively compute the antipath Q in H such that the $d(P_0, Q) \le \varepsilon$.

Suppose this algorithm is called by $Q = DPH(H, \varepsilon)$. This is easy: we decide to go below a node u iff $d(v(u), span(u)) > \varepsilon$. If we do not go below u, we return $Q = span(u)$. Otherwise, we return $Q = DPH(u_L, \varepsilon); DPH(u_R, \varepsilon)$ where u_L, u_R are the left and right children of u.

Extension. We can use this Hierarchy in other ways – as in "view dependent simplifications". Suppose $p \in \mathbb{R}$ is point and we want to simplify the path proportionally to its distance from p. In this case, we can make our decision to go below a node u based on the distance of p from $span(u)$ and $v(u)$.

§4. What is a TIGER Map?

Simplifying real data is much more complex than line simplification. We want to address the TIGER data set. Ken Been's thesis [1] addresss this problem. But first we address a more general question: what really is the mathematical model of a TIGER map? The answer to this question is important if we want to understand simplification of such data. We propose to view the TIGER data as three overlaid geometric structures:

(1) Planar Subdivision Σ: At the crudest level, this amounts to a subdivision of the map area into land and water. This just means that we want to color the plane with 3 colors: blue for water, yellow for land,

and white for non-map areas. We can subdivide the land area into other classifications of interest later (e.g., green or park areas, political boundaries, urban areas, special areas)

(2) Transportation Network Γ: This is a skeleton (i.e., a 1-dimensional network) representing the roads and railway lines.

(3) Landmark Sitemap Λ: This is just a collection of points, representing landmarks. We observe that in the TIGER data, the landmarks are of two types: point landmarks and area landmarks. We will ignore the area landmarks (or replace them by points) for our purposes.

Henceforth, when we talk about a map (TIGER) M, we will mean such a triple $M = (\Sigma, \Gamma, \Lambda)$. The fact that we have three overlaid structures imposes considerable constraints on our map simplification. The first two structures Σ, Γ may share common line features (e.g., a road can serve as a boundary between a park and non-park areas).

You may ask: what is there to simplify in Λ? Well, in the context of zooming, we must have a means to thinning out the set. The simplest method is just to drop landmarks as we zoom out. Alternatively, we can cluster them into "super landmarks" as we zoom out. Actually, such a list can be further generalized as follows: we can associate a **zoom range** with each point landmark. These points are only to be displayed within its zoom range. As an example, suppose certain points represents cities: we will only display a city-point when sufficiently zoomed out, but the city-point will vanish when we are too zoomed out.

§5. Extraction of TIGER Maps

Now that we know what we want (abstractly), we must proceed to extract this data from the real TIGER data. We proceed in two steps:

DECOUPLING STEP: We must first decouple the Σ structure from the Γ structure, as this is mixed up in the TIGER representation.

1. The Σ needs to extract all the polygons (and their boundaries). This can be quite complex processing because of nesting. Recall that Tiger lines are in RT1 and RT2 files while polygons are represented in RTA. The connection between them is in RTI. Basically, we need to join these 4 files. For each polygon, we want a clockwise list of their bounding Tlines. For each Tline in this list, we also need to determine a Boolean flag indicating a forward or backward traversal along the Tline.
2. For the Γ structure, we need to identify those Tlines that is part of this structure. The CFCC code in RT1 will be used for this.
3. For landmarks Λ, we need RT7 which lists all landmarks. We only extract the point landmarks.

INITIAL MERGE STEP: (1) We want to merge polygons with the same type information. What constitute "type"? Assuming that each polygon is part of a geographic entity, we must go to RTC to get the FIPS 55 Class Code and name of the geographic entity. We want to merge all polygons with the same code and same name. Other information may also be needed – for instance, RTS tells you whether a polygon is a water feature (lake, river, sea, etc).

The resulting (merged) polygons is given their own unique identifiers (it is normally inherited from one of its constituent POLYID's.

(2) Similarly, each Tline has a CFCC code and a feature name. Again, we merge all Tlines with the same code and name.

Note that (1) and (2) may now have diverged. In Been's thesis, step (2) is made dependent on the merge in (1) – the rule to merge two lines is simply based on their left- and right-polygons. This results in a simpler model than what we propose. In particular, Been's merging will ensuer that every merged line from (2) is either a boundary between two merged polygons, or is contained inside a unique polygon.

Further Discussion. Call the results of the above merging the **Σ-polygons** and the **Γ-lines**. Among the list of Σ-polygons, we need to determine their inter-relation. Polygons has containment structure. We say polygon P is the parent of another polygon Q if P contains Q. So we want to store parent pointers as well as a list of children polygons. The problem is that the polygon containment hierarchy may have depth more than one. This is illustrated by the following example from Ken's thesis.

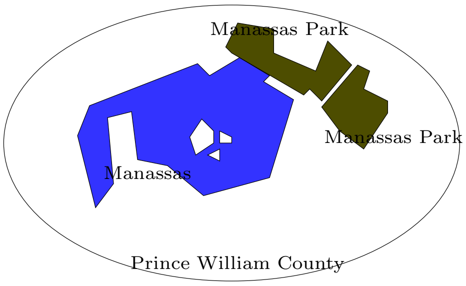

Figure 1: Schematic of Prince William County, Manassas and Manassas Park

Prince William County (PWC) in Virginia encloses two other counties, Manassas and Manassas Park. Moreover, Manassas contains three "islands" of land that belongs to Prince William County.

§6. Preprocessing Details

Let us consider in detail how this can be done in the context of a database system like Postgresql. We will focus here on extracting the geometric information related to nesting of polygons. This is the task:

Let us emphasize that a **polygon** in the following discussion refers to a connected region of the plane. But it need not be simply connected. Hence the boundary of a polygon can have several connected components that we call **loops**. There is a unique loop that is called the **outer loop**. All other loops (if any) are **inner loops**. Two polygons are **adjacent** if they share a common portion of a boundary (sharing isolated points on the boundary do not count). We are given this adjacency information.

The task before us is to determine, for each polygon Q, its **parent**, i.e., the polygon that contains Q. It is possible that Q has no parent. Why is this task necessary? That is because in our map display, it is necessary to draw the parent of a polygon before we draw Q (otherwise the parent will cover up Q when drawn).

Why is this task non-trivial? Because this information may not be directly obtainable from the adjacency information – the parent of polygon Q need not be one of the polygons that are adjacent to Q. This is because of the phenomenon of **clusters**, i.e., a maximally connected set of polygons that are external to each other.

We assume that Tiger information is given to us in the form of two relations (tables):
(i) $Tlines(Tlid, StartPt, EndPt, Detail, LPolyId, RPolyId, \ldots)$

There is an entry for Tiger line. Besides the shape of each line, it tells us the left and right polygons of each Tiger line. The ... indicate other information (mainly to do with the "type information" of the line. This will be important later for simplication.
(ii) $TPolys(PolyId, BoundaryList, \ldots)$

The boundary list is just an unordered list of Tlids, corresponding to all the Tiger lines that bound this polygon.

We shall construct the following intermediate relations in our processing:

- $Loops(LoopId, TlidList, DirectionList, PolyId)$

  The LoopId is a unique identifier we can generate using Postgresql (see below)
- $PolyBoundary(PolyId, Parent, ClusterRep, OuterLoop, InnerLoopList)$

  For each polygon, we want to remember its parent, its outer loop and list of inner loops.

  We also have its cluster information – this is the field *ClusterRep*, which is basically another PolyId. (Clusters are basically represented by a "compressed tree" in the Union-Find data structure.) Each cluster has a **representative** – the *ClusterRep* field is only a pointer to will ultimately lead us to the representative. The *ClusterRep* field of the representative polygon is null – that is how we know we have reached the representative. Initially, each polygon is its own cluster (so its *ClusterRep* field is initially null).

In Postgresql, we can create a "sequence object". This object can give generate a sequence of unique identifiers for use in, say, the LoopIds:

```
=> CREATE SEQUENCE MySeq;
=> INSERT INTO Loops VALUES
(nextval('MySeq'), (123, 456, 789), (true, false, false));
```

The values returned by a sequence object is `BIGINT`. The sequence counter is automatically advanced. If you do not want to advance the sequence counter, just use `currval('MySeq')` instead.

Clustering and Parent Algorithm.

1. The first step is to organize the list of edges on the boundary into loops. To do this, we iterate over each TPolyId. For each boundary list, we partition it into loops. The outer loop is identified.

   Note that this will populate the Loops and PolyBoundary tables described above. The parent of each polygon is itself at this point.
2. For each outer loop, we go through its edges. For each edge in its outer loop, we check if it is also in the outer loop of the adjacent polygon. If so, we "merge" their corresponding clusters.
3. At this point, we know the clusters. We need to set the Parent pointer for the each cluster. This is relatively easy: for each inner loop, find one of the polygons and hence its cluster. The representative of this will have its parent pointer fixed. All the other polygons in its cluster is (implicitly) now pointing to this parent.

§7. Simplification of TIGER Maps

References

[1] K. Been. *Responsive Thinwire Visualization of Large Geographic Datasets*. Ph.d. thesis, Department of Computer Science, New York University, Sept. 2002. Download from `http://cs.nyu.edu/visual/home/pub/`.

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

[3] J. Hershberber and J. Snoeyink. Speeding up the Douglas-Peucker line-simplification algorithm. In *Proc. 5th Intl. Symp. Spatial Data Handling*, pages 134–143, 1992.

[4] J. Hershberber and J. Snoeyink. Cartographic line simplification and polygon CSG formulae in $O(n \log n)$ time. In F. K. H. A. Dehne, A. Rau-Chaplin, J.-R. Sack, and R. Tamassia, editors, *Proc. 5th International Workshop on Algorithms and Data Structures (WADS)*, volume 1272 of *Lecture Notes in Computer Science*, pages 93–103. Springer, 1997.