Visualization, Fall 2001
FINAL PROJECT
DUE: Dec 18, 2001

THIS IS AN UPDATED VERSION.

1 INTRODUCTION

The final project will be a class effort. We will divide the effort into 3 parts.

PART A: Preprocessing the TIGER data into hierarchical LOD structures.

PART B: Constructing Search Structures, on both the server and client sides.

PART C: Constructing the Client/Server System.

We shall aim at working with 16 counties as the goal – these 16 counties will be the 5 boroughs of NYC, plus surrounding counties in New York and New Jersey. Parts and pieces of the necessary code are already embedded in our first 3 homeworks, so reuse them whenever possible. BUT I stres that it is important to stick to the specs and to the deadlines specified below. Any variation in these specs should be discussed with me.

2 PROJECT DETAILS

Our principle data structure will be based on the subdivision datastructure from homework 2. First, we will first review, revise and simplify the `Subdivision class`. Then we will derive from it two subclasses, `Outline class` and `Network class`. These two classes will be the basis for our map representation.

We next describe how to merge and simplify outlines/networks. This will be the basis for constructing a `MergeTree`. This merge tree represents our LOD hierarchy. Finally, we will outline the structure of the client/server architecture, and specify the message protocol.

2.1 SUBDIVISION STRUCTURE

This will be a review and slight revision of the **Subdivision class** in hw2. A subdivision is basically a map. A Subdivision instance has three basic sets: vertices, edges, polygons. The "edges" will be called **Tlines** in our data structure, since there are polygonal lines similar to TIGER Lines. However, to facilitate navigation, each Tline is decomposed into two "halfedges". Corresponding to each Tline, we call the two associated halfedges the **0-halfedge** and the **1-halfedge**. It is arbitrary as to which halfedge is "0" and which is "1".

The main difference from hw2 is a more efficient encoding of these halfedges. Another difference is that we only maintain the successor, but not the predecessor, of halfedges.

We want the subdivision (in fact, all our classes) to be as simple and efficient as possible, and hence avoid the use of private data and special methods for accessing them. Thus all the member variables in the Subdivision class are public.

```
class Subdivision {
  public:
    ////////////////////////////////////////////////////
    // General Subdivision Information
    ////////////////////////////////////////////////////
    int    ID;              // unique ID for this subdivision
    char * name;            // Name or other information
    Point2 southWest;       // south-west corner of bounding box
    Point2 northEast;       // north-east corner of bounding box

    ////////////////////////////////////////////////////
    // Vertex List
    ////////////////////////////////////////////////////
    int numVertices;
    Point2 Vertex[];       // Vertex[i] is a Point2 (from Hill's book)
                           // These are the end points of TLines
                           // Length of Vertex array is numVertices.
    int VertexEdge[];      // VertexEdge[i] is the
                           // index of any TLine incident on Vertex[i]
    ////////////////////////////////////////////////////
    // Polygon List
    ////////////////////////////////////////////////////
    int numPolygons;
    int * PolyEdge; // PolyEdge[i] points to ANY Tline
                    //    that bounds the polygon
    int * PolyType; // 0=unused, 1=water, 2=land, 3=park, 4=landmark
                    //    Color them white/blue/yellow/green/pink.
    ////////////////////////////////////////////////////
    // TLine List  (NOTE: "TLine" is short for TIGER Lines
    ////////////////////////////////////////////////////
    int numTLine;    // Number of TLines
    int * TLineLen;  // LineLen[i] is the number of detailed
                    //     points in the i-th TLine
    Point2 * TLineDetail[];  // LineDetail[i] is the list of detailed
                    //    points in the i-th TLine
    int * TLineType; // 0=isthmus, 1=highway, 2=local road,
```

```
                    //     3=rail, 4=boundary,
                    //     5=river or shoreline, etc
        ////////////////////////////////////////////////////
        // Half-Edge Data
        //        Tline[j] represents two halfedges, which we
        //        call the 0-halfedge[j] and the 1-halfedge[j].
        // Each of the following arrays are of length numTLine
        //   (note that previously, the arrays are of length 2*numLine)
        ////////////////////////////////////////////////////
        int * StartPt;   // StartPt[j] is the (index of the)
                    //    start vertex of 0-halfedge[j], and thus
                    //    the end vertex of the 1-halfedge[j]
        int * EndPt;    // EndPt[j] is the end point of 0-halfedge[j]
        int * Succ0;    // Succ0[j] is the index of the Tline that is
        int * Succ1;    //    the successor of the 0-halfedge[j]
        bool * SuccHe0; // If SuccHe0[j]=x, it means that the successor
                    //    of the 0-halfedge[j] is x-halfedge[j].
        bool * SuccHe1; // Analogous to SuccHe0, but referring to
                    //    to the successor of the 1-halfedge
        int * Poly0;    // Poly0[j] is the polygon bounding the
        int * Poly1;    //    0-th halfedge.  CONVENTION: following
                    //    the Succ halfedges will go CW about
                    //    this polygon.
                    // NOTE: we no longer keep the predecessor
                    //    halfedge links
    }//class Subdivision
```

2.2 OUTLINE/NETWORK LAYERS and their MERGING/SIMPLIFYING

In hw2, we have a class `TigerSubdiv` with two methods `TigerSubdiv::extractOutline` and `TigerSubdiv::extractRoads`. The first method returns a true subdivision, but the second method returns only a **skeleton**. A skeleton is a 1-dimensional complex with only a list of vertices and a list of TLines, together with their associated halfedges; the polygon information is omitted. Hence we can continue to use a TigerSubdiv data structure to represent skeletons. You should think of the results of these two extractions as the initial data for all subsequent simplifications and merging. They are henceforth INDEPENDENT of each other. This may lead to incorrect relative geometry. E.g., a simplified road might unexpectedly pass through a simplied coastline into the sea. But we tolerate this kind of error because of our pixel-based error bounds (so that they are usually not very visible). Call these two data the **outline layer** and the **network layer**.

We simply derive the Outline and Network classes from Subdivision, by adding some special methods:

```
////////////////////////////////////////////////////
// Derived classes for Outline and Network
////////////////////////////////////////////////////

class Outline : public Subdivision {
  public:
    Outline merge( Outline ol2 );  // merge *this outline with ow2
```

```
    Outline simplify( );
    void write( char * filename );
    void read( char * filename );
}

class Network : public Subdivision {
  public:
    Network merge( Network nw2 );  // merge *this network with nw2
    Network simplify( );
    void write( char * filename );
    void read( char * filename );
```

We need to store and read these structures to and from files, hence the read/write methods. The merge/simplify methods are to support the construction of a LOD hierarchy (see next subsection).

We describe the merge and simplify methods for these two layers of data. As noted, the simplification and merging of these two layers are now completely autonomous relative to each other. Merging is relatively simple: there are three steps:

- First, we form the union of the lists of vertices, Tlines, etc.
- Then we have to remove duplicate values (at the boundary of the two subdivisions).
- Finally, we need to do actual merging. This depends on whether we are merging Outline or Network. For Network, we first find all those end points (Vertices) of degree 2. The two Tlines incident on such a vertex is merged, and the Vertex removed (but converted into a detail point of the merged Tline).
- For Outline, We check each Tline to see the bounding polygons on both side. If these 2 polygons are the same type, BUT distinct, then we remove this Tline, and merge the two distinct polygons. NOTE that if the 2 polygons are identical, then this Tline is an isthmus, and we do not remove it.

Simplification is based on some maximum error bounds denoted `sScale` which, as in hw2, means the "screen scale". Thus `sScale = 87` means 87 meters per pixel. There are two steps: (1) First, we simplify TLines, using the the same Douglas Peucker algorithm as specified in hw2. (2) Next, we eliminate small polygons. The rule here is that for any polygon whose bounding box area is less than `MIN_PIXEL_AREA` pixels will be eliminated. We recommend the value `MIN_PIXEL_AREA=8`. Actually, we will apply this heuristic only to the Outline Simplification. For Network simiplication, since there are no polygons, we use a slightly different heuristic: when a Tline is shorter than `MIN_PIXEL_LENGTH` pixels, we collapse that Tline. We recommend the value `MIN_PIXEL_LENGTH=3`.

```
////////////////////////////////////////////////////
// methods to reomve polygons or Tlines
////////////////////////////////////////////////////

void Outline::checkPolygon(int polyID, int sScale)
// Check if polyID has area that is too small
// Remove it if so.

void Outline::removePolygon(int sScale)
// Apply checkPolygon to *this Outline.
```

```
void Network::checkTline(int tlineID, int sScale)
// Check if tlineID has length that is too small
// Remove it if so.

void Outline::removeTline(int sScale)
// Apply checkTline to *this Network.
```

2.3 MERGETREE LOD HIERARCHY

We now describe the LOD Structure. This is a full binary tree called the `MergeTree`. "Full" means that each internal node of this binary tree has two children. The leaves of the MergeTree represent individual counties. Each internal node represents a simplification of the information found in its two children.

Our target for this final project is a merge tree for 16 counties. These 16 counties should include the 5 boroughs of New York City. Furthermore, they should form a simply connected geographical region, including some portions of New Jersey. Because we have only a small merge tree, we want you to design the merge tree by hand.

NOTE: See http://cs.nyu.edu/ yap/classes/visual/01f/hw/finalProj/counties16.html for a list of 16 counties proposed by Ming-feng.

What are the constraints in forming the merge tree? There is only one hard constraint: the **map coverage** of each node should be connected. The only exception is at the leaves where we may have counties that are NOT connected (e.g. New York County a.k.a. Manhattan is not connected because it owns the Statue of Liberty Island which is completely surrounded by waters belonging to Hudson County in New Jersey). This means that at the next level, New York County MUST be merged with Hudson County to ensure that the next level is connected. Actually, this does not guarantee that every node in the next level has a connected coverage – but as far as we know, there are no other possibilities in the TIGER map. By the way, this example shows that Hudson County is not simply-connected (though it is connected).

There are other soft constraints like: you should try to make map coverage of each node have large area/perimeter ratios, thus eschewing long narrow coverage. Another soft constraint is to make the aspect ratio of the bounding box as close to 1 as possible. But how do you satisfy the hard constraint? You need to know the adjacency relation between counties! Two counties are adjacent if they share a common border. We have a precomputed database for this information. Under our TIGER resource page (http://cs.nyu.edu/visual/home/proj/tiger/) there is a list of 435 counties from 14 states in North East USA, from Maine to Virginia. This list contains the adjacency graph of these counties.

We must now describe what is stored in each node of the MergeTree. The map coverage of each node is represented by the Outline and Network layers for the maps. We will be using the MergeTree structure for answering range queries. Hence at each node, we need a range query search structure to answer range queries for its Outline and Network layers. This is captured by the following class definition:

```
////////////////////////////////////////////////////
//        Merge Node class
////////////////////////////////////////////////////

class MergeNode {
```

```
public:
      int ID;                // index to MergeNodeArray
      int xmin, xmax;        // bounding box
      int ymin, ymax;        //       information
      int sScale;            // Level of Detail (LOD)
      int leftChild, rightChild; // indices of children
                             //       (-1 if no such child)
      Outline   oStruct;     // Outline layer
      Network   nStruct;     // Network layer
      RangeTree rStruct;     // range tree search structure
      HashTable hStruct[];   // hash table array: hStruct[j] is the
                             //       hash table for the j-th client

// CONSTRUCTORS:

      MergeNode(Subdiv & ts);  // create a leaf node from a
                      // subdivision structure
      MergeNode(MergeNode & mn1, MergeNode & mn2);
                      // create an internal node from two children.
                      // It should call the merge/simplify methods
                      // of the Outline and Network classes

// METHODS:

      void Dump(char * fname);  // Write out to one or more files

} //class MergeNode
```

2.4 MERGETREE QUERYING

How we do perform a window query at a MergeNode? The following method is used The input for the query is

```
void MergeNode::query(Rect q, int sScale, int clientID)
```

where q is the query window (represented by its southwest and northeast corner points), sScale is the screen scale value, and clientID is an identifier for the client. Note that sScale=170 means that each pixel width is 170 meters.

We start at the root and recursively pass the parameters (q, sScale, clientID) to the children when appropriate. Suppose we are at some MergeNode. First verify that the window q intersects the bounding box at the present MergeNode. If not, we return nothing and quit. Otherwise, there are two cases.

CASE 1: The sScale at the current MergeNode is greater than the sScale of the query. Then the window search is recursively sent to the children of the current MergeNode.

CASE 2: The sScale at the current MergeNode is less than or equal to the sScale of the query. In this case we directly search the current node, as follows. We first use the range tree to retrieve all TLines with

at least one endpoint within the query window q. These TLines are given as two lists $TListOutline$ and $TListNetwork$, one for the Outline and another for the Network object.

At this point, we have to consider further processing of these two lists, to decide exactly what to send to the client. This processing depends on the particular client.

(A) Processing $TListNetwork$: For each TLine in this list, we Each TLine has a unique ID. We try to insert these TLines into the hash table. If unsuccessful, it means that the client already has this information. Otherwise, this is sent to the client. Similarly, we retrieve all polygons in the Outline structure, try to insert them in the hash table. Again, only send if the insertion is successful.

This completes the duty on the server side. What happens at the client side? The client also has the merge tree, but does not need the hash table. Also, its Outline and Network and Range structures are initially empty. As vertices, Tlines and polygons are sent, we fill them into the structure.

We need to worry about main memory: it is assumed that each MergeNode data is stored in a set of files and they are read into main memory as needed. We need to do a little bit of simple paging.

```
////////////////////////////////////////////////////
// Array of Merge Nodes, organized as a complete
//   binary tree (node i has children 2i and 2i+1
////////////////////////////////////////////////////

MergeNode MergeNodeArray[ NumMergeNodes ];
        // If there are 16 leaves, NumMergeNodes = 31.
```

The RangeTree will contain all the endpoints of TLines in these two classes to support orthogonal range queries.

We will assume the the RangeTree stores the following points with associated data: for each TLine L in Outline and Network object, we store

$$(x, y, L, flag), (x', y', L, flag)$$

where (x, y) and (x', y') are the endpoints of L (and L is really the TLine ID in the Outline or Network object). Given a window q as query, this RangeTree will return two lists of TLine IDs, one list for the Outline object, and one list fro the Network object.

2.5 CLIENT-SERVER ARCHITECTURE

We assume that the server forks to serve each client and the messages between them are as follows:

1. INIT(x,y): client sends this message to begin the visualization at position (x,y).
2. INITDATA: server sends the initial data to begin the visualization.

3. WIN(xmin, xmax, ymin, ymax, sScale): client requests a data in a query window, at a given level (=sScale) of detail.

4. DATA: response to WIN(xmin, ...) request. Note that in general, we can have several responses to a single request. Below, it is assumed that there will be two responses – one for the Outline data, one for the Network data.

5. TERMINATE: client requests termination.

More details of these messages will be given below. The client will have four threads, each with its own input queue ("Q").

1. Input Thread: reads from the mouse and keyboard. It puts data into the NetworkQ or into the DisplayQ. The commands it accepts are PAN, ZOOM, JUMP, PRIORITIZE, QUIT.

2. Network Thread: It processes the NetworkQ requests converts this into WIN(xmin, ymin, xmax, ymax, level) requests. At the same time, it puts a copy of this request into the DataQ.

3. Data Thread: As soon as it read new data from the network, it compares them to the WIN requests. After decoding this, it informs the DisplayQ.

4. Display Thread: it updates the display as requests and new data arrives.

2.6 MESSAGE PROTOCOL

Let us now specify the data that are sent in the above messages.

First we give a description of what we ultimately want, because what we ask you to implement here is a simplified version. We want to client to reconstruct a stripped down version of MergeTree on its side. Each MergeNode on the client side should have an Outline and Network, of course. But we also need a RangeTree to answer local queries. The only thing we do not need is the HashTable array.

Now, this RangeTree could be semi-dynamically built up as Vertices are sent across the thinwire. However, semidynamic range trees are hard to implement and do not seem practical. So we believe the correct structure to build is a partial copy of the *static* range tree that is on the server side. This means that we have to send substructures corresponding to this static range tree, in addition to the Outline/Network data. But in this final project, we will skip this detail. We will assume that you implement some cheap semi-dynamic structure for range queries. Initially, this can be a brute force search through the date structure. But we believe you can cut down this cost substantially by a simple bucketing technique – physically subdivide the bounding box of the map coverage into some number of rectangles (say, 16). Each rectangle has a list ("bucket") that stores all the query-able points in that rectangle.

We are now ready to specify the messages to be communicated. The two non-trivial messages are INITDATA and DATA.

0.	INITDATA	
1.	int N = 16;	
	// 16 is the number of counties	
		// Assume N to be a power of 2
2.	(// Following is a per mergeNode basis	
	// There are (2*N - 1) = 31 mergerNodes	
2.1	int xmin, ymin;	// SouthWest corner of bounding box.
2.2	int xmax, ymax;	// NorthEast corner
2.3	int sScale;	// screen scale
	// OUTLINE INFORMATION	
2.4	int numVertices	
2.5	int numEdges	
2.6	int numPoly	
	// NETWORK INFORMATION	
2.7	int numVertices	
2.8	int numEdges	
	) // end of per mergeNode data	

Based on the above information, the client can now set up "empty arrays" to receive data as they arrive. However, if the arrays are too large, we may need some virtual arrays. See below for one solution.

	DATA	
1.	bool OutlineOrNetwork=0	// 0=Outline data, 1=Network data
2.	int MergeNodeID	
3.	int numVertices	// number of vertices being sent
4.	int numTlines	// number of Tlines being sent
5.	int numPoly	// number of polygons being sent
	(// per vertex data	
6.1	int vertexID	
6.2	point2 p	
	)* // end of per vertex data	
	(// per Tline data	
7.1	int tlineID	
7.2	int startPt, endPt	// vertex indices of endpoints
7.3	int succ0, succ1	// Tline indices of 0- and 1-successor
7.4	bool succ0he, succ1he	// halfedge bits of 0- and 1-successor
7.5	int leftPolyID, rightPolyID	// bounding polygons to teh left and right
7.6	int numDetail	// number of detail points
7.7	point2 pointList[]	// array of detail points
	)* // end of per vertex data	
	(// per polygon data	
8.1	int polyID	
8.2	int polyType	// type of polygon (land, water, etc)
8.3	int polyTlineID	// any bounding Tline of this polygon
	)* // end of per polygon data	

There are two kinds of DATA messages. This is indicated by the very first value which is a boolean: 0 means Outline data and 1 means Network data. The above, we describe the Outline data. But the Network data message is just a subset of the Outline data message: we simply omit the information related to polygons. Specifically, the following will be omitted: 5, 7.5, 8.1, 8.2, 8.3.

2.7 ADVANCED TOPICS

This section contain advance topics that you may consider implementing.

2.8 Automatic Merge Tree Construction

Above, we asked you to design the MergeTree structure by hand because it has only 16 counties. In general, we would want to design an algorithm subject to hard constraint and various soft constraints. See the TIGER Resource webpage for more information on this topic.

2.9 Partial Range Search Trees

The RangeTree structure on the client side should be a partial copy of the RangeTree on the server side. Each time the server sends new Outline/Network data to the client, it must also send information for augmenting the Partial RangeTree.

2.10 Virtual Arrays

The client may not have as much memory as we would like to think. In particular, it may be too much to preallocate all the empty arrays. What we want are "virtual arrays", which we can index in the normal way. This can be done by a generalization of the notion of dynamic tables. Let N be the "virtual size" of the array, and $0 \leq n \leq N$ be the number of non-empty entries. Initially, $n = 0$.

The goals are: (0) We are able to access any array entry by indexing as in regular arrays. But we can also test if an entry is "empty" or "used". Initially all entries are empty, and deletion makes an entry empty. Initiallization is $O(1)$ time. (1) The space usage is $O(n)$ where n is the number of used entries. Thus space is independent of N. (2) When the array is full, the speed of accessing this array should be as fast as a normal array. This assumes that an array of size N will fit in the main memory. (3) The amortized cost of inserting and deleting an entry is $O(1)$.

For our purposes, we can dispense with deletions. We may want to augment the operations to allow a "block insertion or deletion".

THE END