TGR2KML Whitepaper

Bruce A. Ralston
University of Tennessee
bralston@utk.edu

October, 2007

The purpose of this white paper is to describe the structure and the logic used in TGR2KML. It does not explain every line of code, nor is it meant to be a detailed description of TIGER file contents or how to use TIGER files with Census 2000 data. For information on using TIGER and Census 2000 files, I humbly suggest you read <u>GIS and Public Data</u>, written by this author and available from OnWord Press. This paper is meant give those interested in developing software for GIS an idea of how TGR2KML is structured. In working with students, and in my own self-education, I have found some of the ideas used in TGR2KML, particularly efficient extracting of the topology in TIGER and the strategy for quickly assembling area features, to be quite instructive. Students who have been exposed to these ideas seem to have a better understanding of how programs can be structured to exploit topological relationships.

TGR2KML is based on TGR2SHP. However, TGR2KML extracts only area features, not lines or points. Readers interested in how those features can be processed should see the TGR2SHP whitepaper. I have left the code for extracting point and line features in the source files. If there is a need for these in KML format, I will add that in, or perhaps an interested user will add them in. Readers who have read the TGR2SHP whitepaper and are only interested in how the KML files are written can proceed to section entitled Writing KMLs.

Some Comments on the Software

The program is written in version 6 of Microsoft Visual C++. While written in C++, it is really a C program. That is, I have not made any attempt to introduce classes, take advantage of late binding, or use inheritance. I do make a good deal of use of standard C practices such as user defined structures, dynamic memory allocation, and visual interface programming tools.

There is one piece of third-party commercial code used in the program. Xceed Software's dll for unzipping files is referenced in the data input sections of the code. The redistributable version of that dll is included in the TGR2KML installation program. However, if you wish to compile the code yourself, you will have to do one of three things:

1. Get your own developer's version of Xceed Zip
2. Use a different product and replace the Xceed Zip calls with that of the component you choose
3. Comment out all references to zip files and process only uncompressed TIGER files

A Little History

In the mid-1990s a group of Political Science students at the University of Tennessee asked for my help in geocoding automobile thefts using TIGER files and ArcINFO. I asked each several questions: Do you have a UNIX account? Have you ever used UNIX? Have you ever used ArcINFO? Have you ever used the TIGERTools AMLs? As I watched each student struggle with simply getting a useable database, I thought "There has got to be a better way." Not finding one that I could afford, I set out to write my own

TIGER translator. The result was TGR2SHP. In the August of 2007, I extended the program to generate KML files.

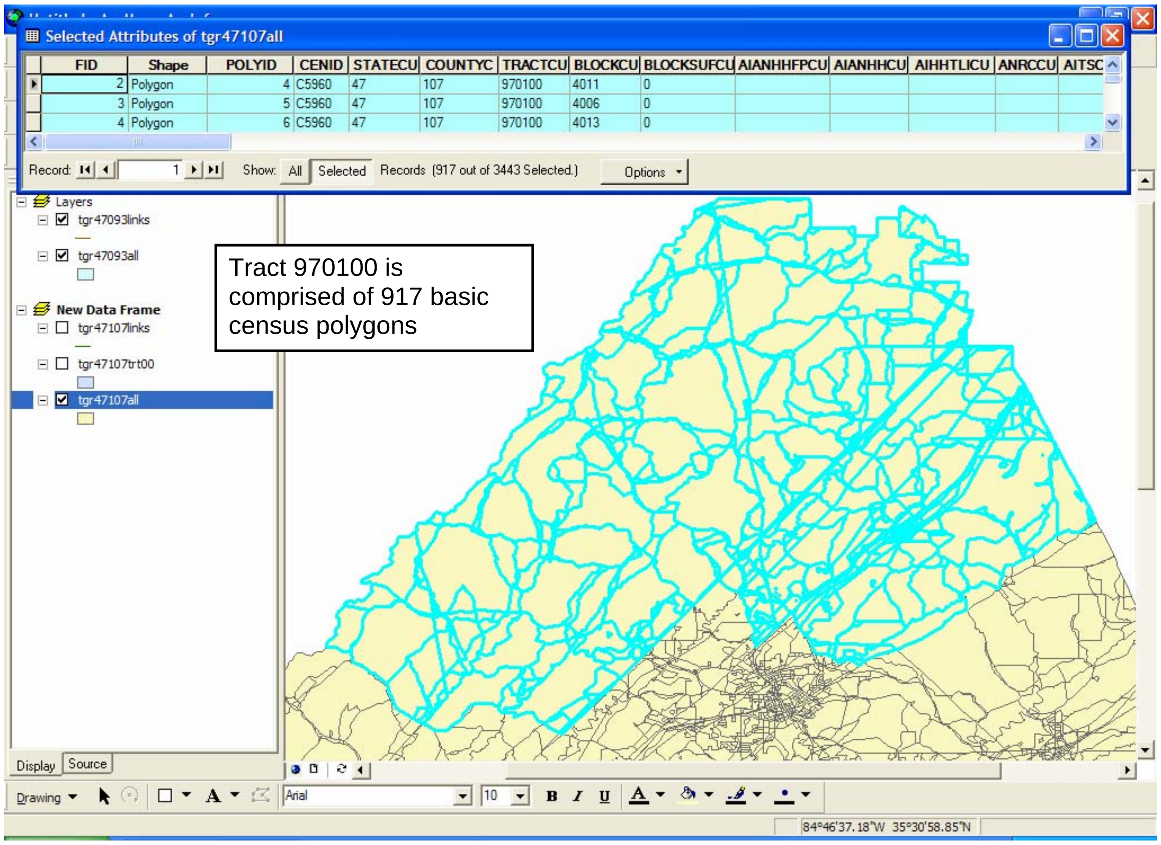
Program Overview

TGR2KML consists of three compiled modules. The first module is the executable TGR2KML.EXE. This module is used to gather user inputs and to manage calls to one of two dynamic linked libraries. The two dlls correspond to the version of TIGER being processed. TGRSHP.DLL is used to process TIGER97 through 108th Congressional Districts TIGER. TGRSHP2.DLL is used to process TIGER 2002 through TIGER 2005, Second Edition files. These are the most current files at the time of this writing. Each dll exposes one function, either Display (TGRSHP.DLL) or Display2 (TGRSHP2.DLL). The functions take similar arguments: the name of the TIGER file (including its path), the path to the output directory, a string containing the layers and options the user wishes to use, the sequence number of the current input file and the total number of files to be processed.

TIGER Overview

A basic understanding of TIGER files is necessary before looking at the program code. A county's TIGER file consists of a set of files with each file containing a specific record type. The record types, which contain specific types of information, are described in detail in the Census Bureau's TIGER/Line file technical documentation. The most recent version of that document can be found at http://www.census.gov/geo/www/tiger/tiger2006se/TGR06SE.pdf. Here I will just highlight certain TIGER concepts and record types that will be discussed later in this document.

- Each line in a TIGER file is assigned a TIGER/Line ID, Permanent 1-Cell Number. This is a unique identifier assigned to every line in the TIGER archive. TGR2SHP often uses TLIDs to connect information from one record type to corresponding information in another record type.
- TIGER files follow strict planar enforcement. That is, where lines cross, a node is created and a set of lines that encloses an area defines a basic census polygon, also referred to as GT-polygons (Figure 1). Basic polygons are topological constructs resulting from planar enforcement. Every basic census polygon in a TIGER file is assigned Census ID (CENID) and polygon ID (POLYID). Together these numbers give each polygon a unique identifier.



Selected Attributes of tgr47107all

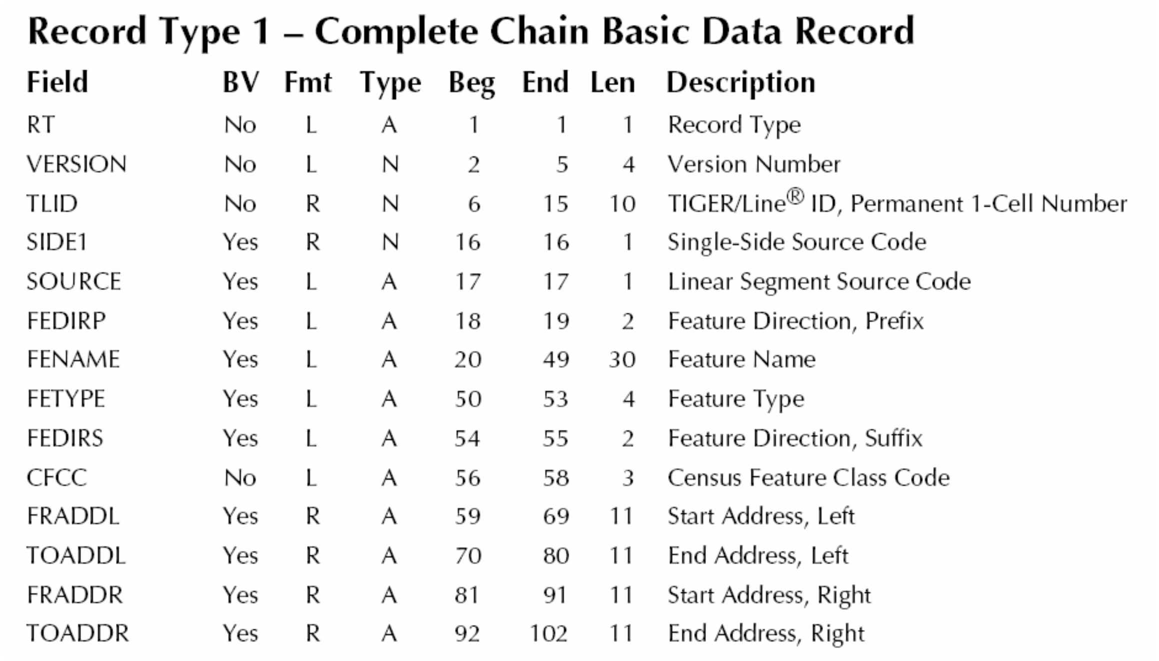
FID	Shape	POLYID	CENID	STATECU	COUNTYC	TRACTCU	BLOCKCU	BLOCKSUFCU	AIANHHFPCU	AIANHHCU	AIHHTLICU	ANRCCU	AITSC
2	Polygon	4	C5960	47	107	970100	4011	0					
3	Polygon	5	C5960	47	107	970100	4006	0					
4	Polygon	6	C5960	47	107	970100	4013	0					

Record: 1 Show: All Selected Records (917 out of 3443 Selected.) Options

Tract 970100 is comprised of 917 basic census polygons

Figure 1. The Basic Census, or GT, Polygons that define a census tract are highlighted. Note the Polyid and Cenid fields in the attribute table.

- Record Type 1 – Complete Chain Basic Data Record (RT1) contains a TLID, the beginning and ending coordinates (Latitude/Longitude) of the line, and information often associated with geocoding, such as street name, type, and address ranges. It also contains the Census Feature Classification Code (CFCC) which indicates the type of line it is (road, rail, waterway, political boundary, etc.) There is other information associated with RT1, but these are the attributes discussed in this document (Figure 2).

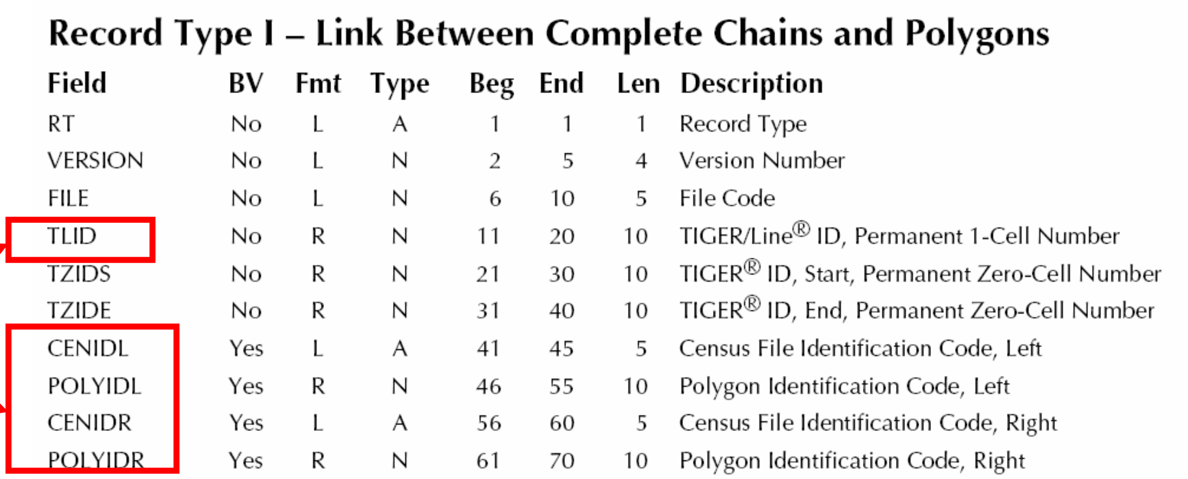


Record Type 1 – Complete Chain Basic Data Record

Field	BV	Fmt	Type	Beg	End	Len	Description
RT	No	L	A	1	1	1	Record Type
VERSION	No	L	N	2	5	4	Version Number
TLID	No	R	N	6	15	10	TIGER/Line® ID, Permanent 1-Cell Number
SIDE1	Yes	R	N	16	16	1	Single-Side Source Code
SOURCE	Yes	L	A	17	17	1	Linear Segment Source Code
FEDIRP	Yes	L	A	18	19	2	Feature Direction, Prefix
FENAME	Yes	L	A	20	49	30	Feature Name
FETYPE	Yes	L	A	50	53	4	Feature Type
FEDIRS	Yes	L	A	54	55	2	Feature Direction, Suffix
CFCC	No	L	A	56	58	3	Census Feature Class Code
FRADDL	Yes	R	A	59	69	11	Start Address, Left
TOADDL	Yes	R	A	70	80	11	End Address, Left
FRADDR	Yes	R	A	81	91	11	Start Address, Right
TOADDR	Yes	R	A	92	102	11	End Address, Right

Figure 2. Part of Record Type 1 Technical Description. Source: TIGER/Line Files Technical Documentation, US Bureau of the Census.

- Every line in a county's TIGER Record Type 2 – Complete Chain Shape Coordinates (RT2) contains a TLID and all the shape points necessary to capture the shape of each line. There is a Zero-to-Many relationship between RT1 and RT2. That is, a straight line segment will have no entries in RT2 (its start and end points are in RT1). A line with curves will have shape points, and thus one or more entries in RT2.
- Record Type I (RTI) contains one record for every line. Each record indicates the basic census polygon on the left and right of the line. By themselves, basic polygons may or may not have any meaning with respect to other data sets, such as the Census of Population. In more recent versions of TIGER RTI also indicates the From Node and To Node ids for each line. The key point here is that RTI allows one to associate lines with basic census polygons (Figure 3).



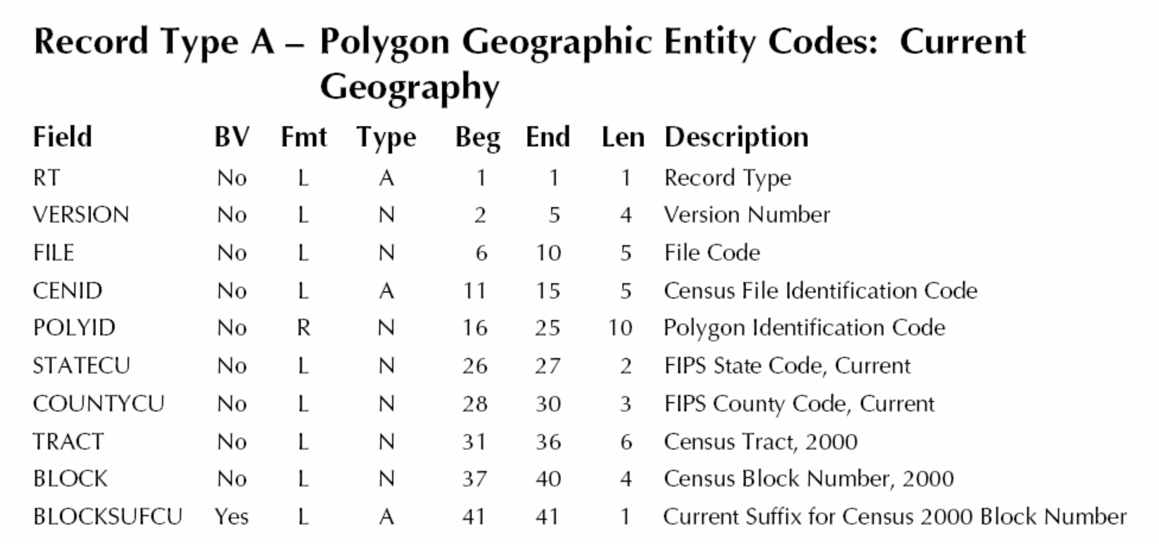
Record Type I – Link Between Complete Chains and Polygons

Field	BV	Fmt	Type	Beg	End	Len	Description
RT	No	L	A	1	1	1	Record Type
VERSION	No	L	N	2	5	4	Version Number
FILE	No	L	N	6	10	5	File Code
TLID	No	R	N	11	20	10	TIGER/Line® ID, Permanent 1-Cell Number
TZIDS	No	R	N	21	30	10	TIGER® ID, Start, Permanent Zero-Cell Number
TZIDE	No	R	N	31	40	10	TIGER® ID, End, Permanent Zero-Cell Number
CENIDL	Yes	L	A	41	45	5	Census File Identification Code, Left
POLYIDL	Yes	R	N	46	55	10	Polygon Identification Code, Left
CENIDR	Yes	L	A	56	60	5	Census File Identification Code, Right
POLYIDR	Yes	R	N	61	70	10	Polygon Identification Code, Right

Figure 3. RTI relates lines to polygons. This figure shows part of the RTI definition. The red boxes

were added by the author. Source: TIGER/Line Files Technical Documentation, US Bureau of the Census.

- Record Type A (RTA) contains information relating basic census polygons to polygon entities that have more than a topological meaning, such as Census Tracts, Counties, Blocks, American Indian/Alaska Native/Hawaiian Homeland Areas, and the like (Figure 4). The entities in RTA refer to the current geography, where current refers to the TIGER version's year. Thus, for TIGER 2005 second edition files, the Census Tracts referenced in RTA would relate to the tracts as defined in 2005.

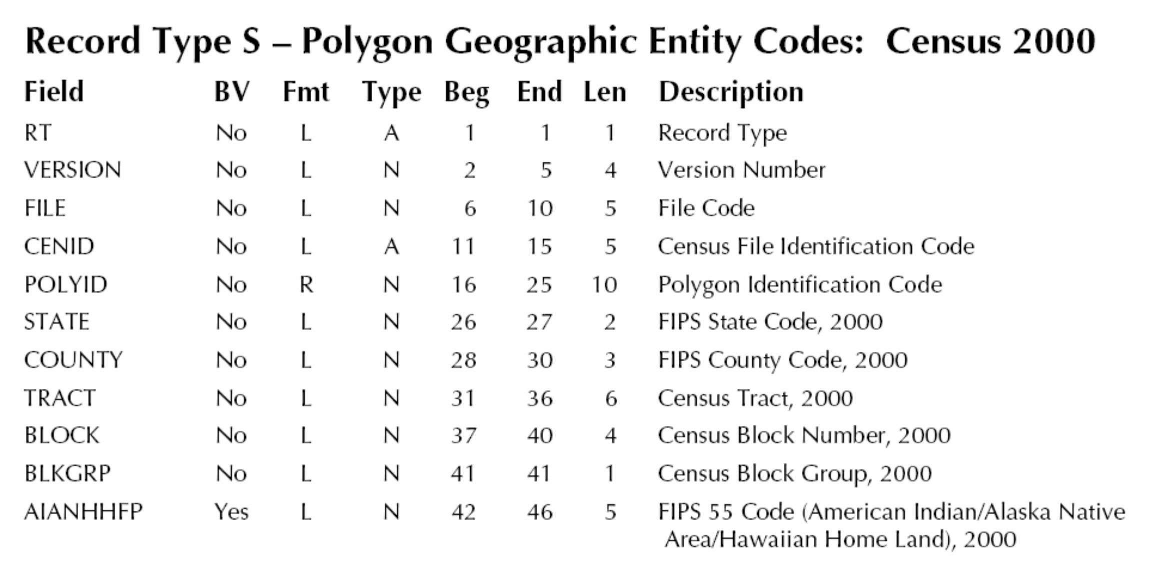


Record Type A – Polygon Geographic Entity Codes: Current Geography

Field	BV	Fmt	Type	Beg	End	Len	Description
RT	No	L	A	1	1	1	Record Type
VERSION	No	L	N	2	5	4	Version Number
FILE	No	L	N	6	10	5	File Code
CENID	No	L	A	11	15	5	Census File Identification Code
POLYID	No	R	N	16	25	10	Polygon Identification Code
STATECU	No	L	N	26	27	2	FIPS State Code, Current
COUNTYCU	No	L	N	28	30	3	FIPS County Code, Current
TRACT	No	L	N	31	36	6	Census Tract, 2000
BLOCK	No	L	N	37	40	4	Census Block Number, 2000
BLOCKSUFCU	Yes	L	A	41	41	1	Current Suffix for Census 2000 Block Number

Figure 4. Record Type A relates basic polygons (CENID and POLYID) to current geography polygon entities such as counties, tracts, and blocks. This figure shows part of the RTA definition. Source: TIGER/Line Files Technical Documentation, US Bureau of the Census.

- Record Type S (RTS) is similar to RTA, except that the polygons referenced in it refer to the definitions of areas at a particular census year. Thus, for TIGER 2005 second edition files, the Census Tracts referenced in RTS would relate to the tracts as defined for the 2000 Census of Population (Figure 5)



Record Type S – Polygon Geographic Entity Codes: Census 2000

Field	BV	Fmt	Type	Beg	End	Len	Description
RT	No	L	A	1	1	1	Record Type
VERSION	No	L	N	2	5	4	Version Number
FILE	No	L	N	6	10	5	File Code
CENID	No	L	A	11	15	5	Census File Identification Code
POLYID	No	R	N	16	25	10	Polygon Identification Code
STATE	No	L	N	26	27	2	FIPS State Code, 2000
COUNTY	No	L	N	28	30	3	FIPS County Code, 2000
TRACT	No	L	N	31	36	6	Census Tract, 2000
BLOCK	No	L	N	37	40	4	Census Block Number, 2000
BLKGRP	No	L	N	41	41	1	Census Block Group, 2000
AIANHHFP	Yes	L	N	42	46	5	FIPS 55 Code (American Indian/Alaska Native Area/Hawaiian Home Land), 2000

Figure 5. RTS contains information similar to that in RTA, except the geography is for the Census 2000. This figure shows part of the RTS definition. Source: TIGER/Line Files Technical Documentation, US Bureau of the Census.

- Record Type C (RTC) relates names of areas, such as county name, town name, or school zone name, to various census entities.

There is much more information in TIGER files and TGR2KML does read and use much of that information. For information on the contents of TIGER files, the interested reader is referred to the TIGER/Line file technical documentation referenced earlier in this document. The above points are meant to give the reader enough information to follow the program descriptions below.

A Quick View of the EXE

TGR2KML.EXE is used to collect user inputs (the TIGER files to process, their version, the layers to extract, and the output options) and to make calls to the proper DLL, either TGRSHP.DLL or TGRSHP2.DLL.

The main dialog (Figure 6) allows users to specify the TIGER files to be processed. Options exist for deleting files, adding more files, editing previously selected layer options, and executing or canceling the program.

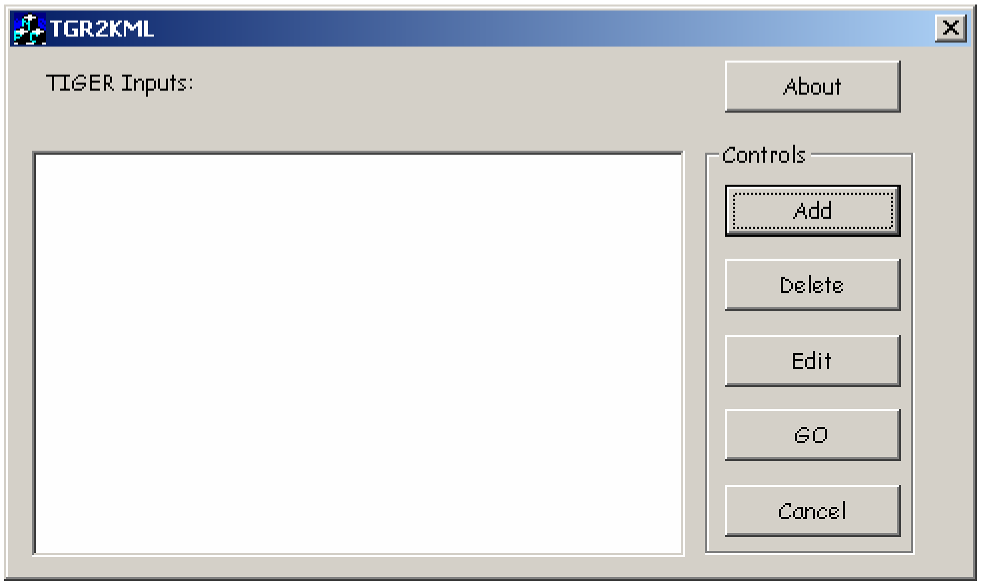
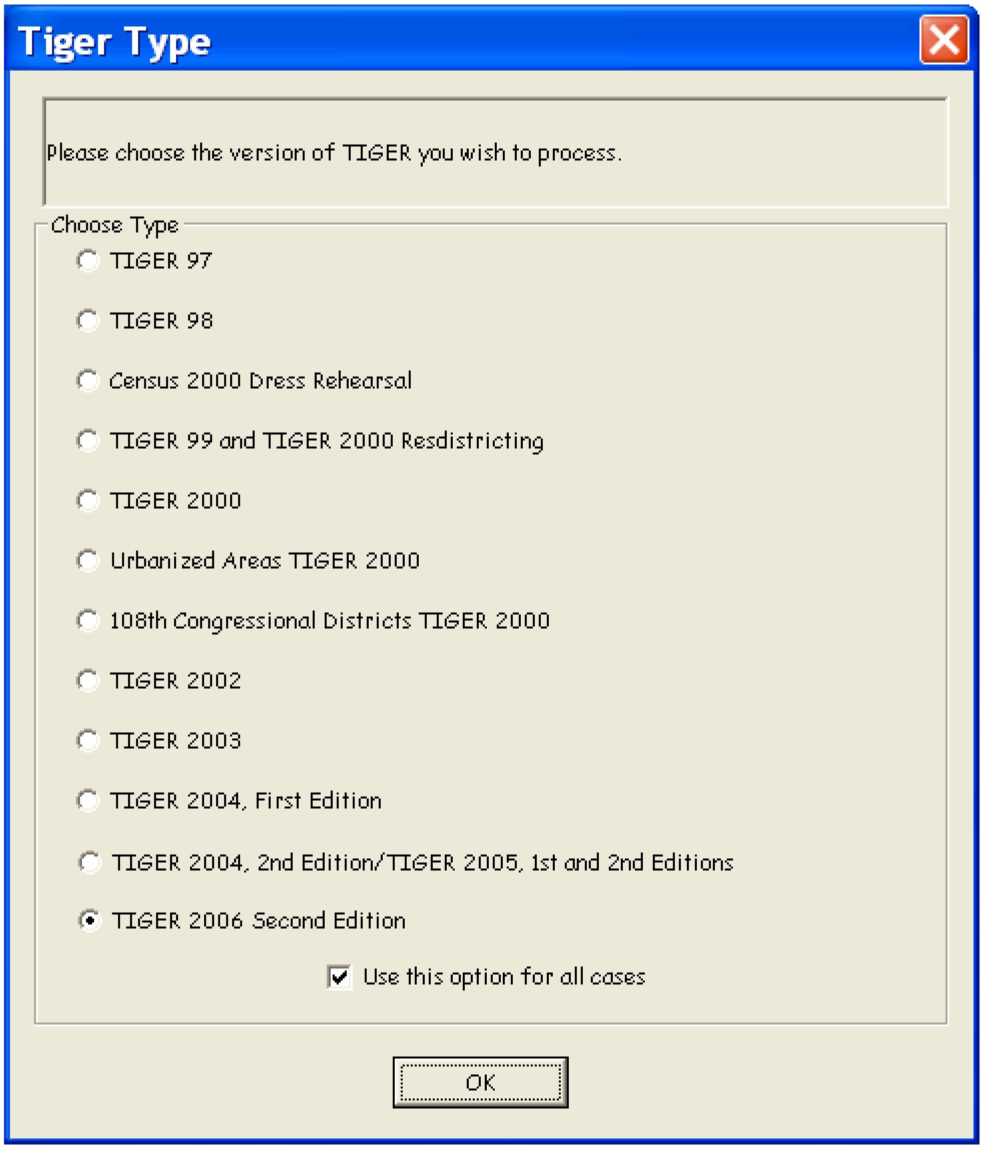


Figure 6. TGR2KML main dialog. The code corresponding to this dialog is in tgrshpdlg.cpp.

Once a TIGER file is selected, the user is asked to specify the type of TIGER archive (if it is 1997 TIGER or later). See Figure 7. Earlier versions of TIGER had unique file extension which made it possible to determine the TIGER version from the file name. Since 1997, all TIGER files have the same file extensions, hence the need for this dialog. The user's choice is stored in an integer variable named CurrentVersion.



Please choose the version of TIGER you wish to process.

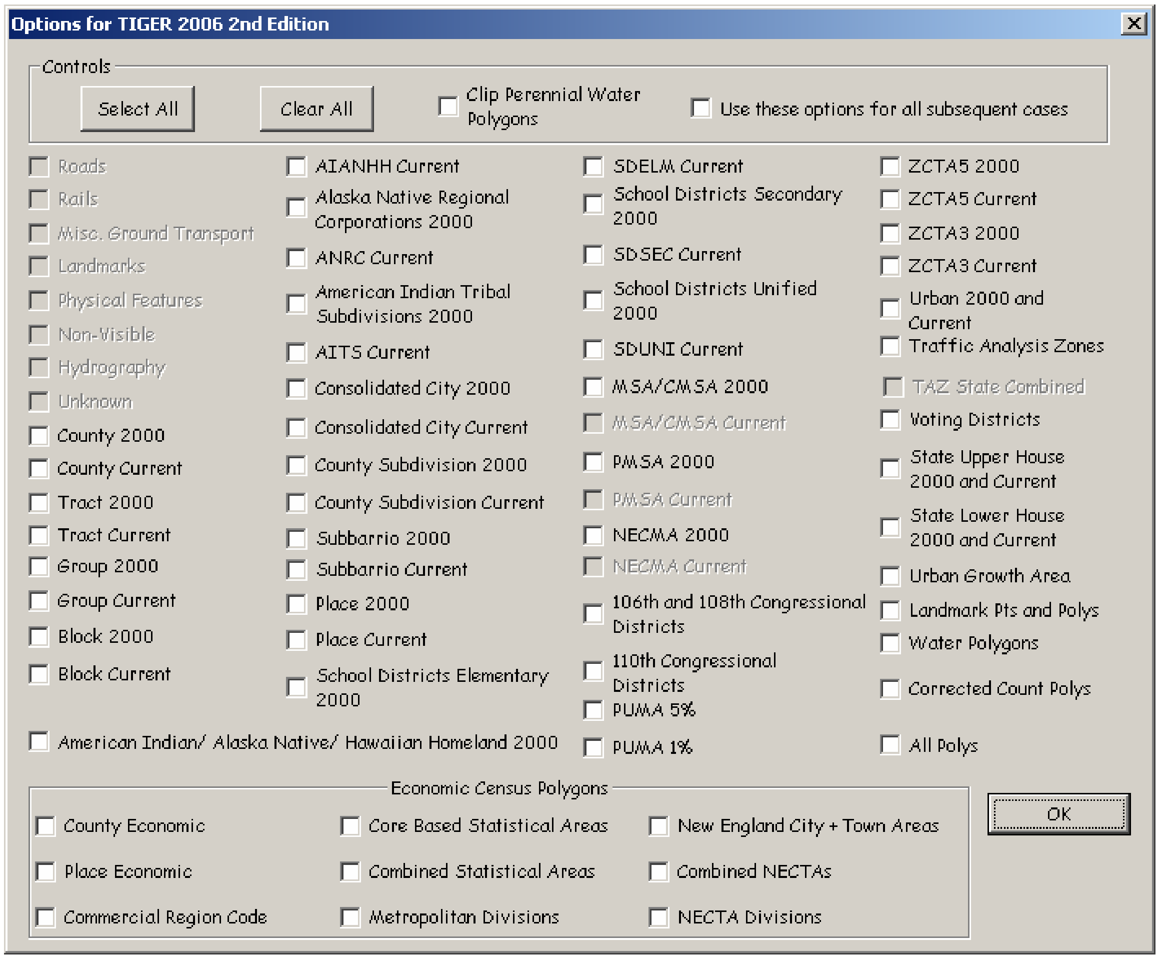
Choose Type

- ( ) TIGER 97
- ( ) TIGER 98
- ( ) Census 2000 Dress Rehearsal
- ( ) TIGER 99 and TIGER 2000 Resdistricting
- ( ) TIGER 2000
- ( ) Urbanized Areas TIGER 2000
- ( ) 108th Congressional Districts TIGER 2000
- ( ) TIGER 2002
- ( ) TIGER 2003
- ( ) TIGER 2004, First Edition
- ( ) TIGER 2004, 2nd Edition/TIGER 2005, 1st and 2nd Editions
- [✓ TIGER 2006 Second Edition]

[✓] Use this option for all cases

Figure 7. Tiger Type dialog. The code corresponding to this dialog is in tigertypedlg.cpp

Knowledge of the TIGER version is necessary to present the user the proper layer options (Figure 8). That is, the layer choices vary depending on the version of TIGER file being processed. The check box values (1 or 0) are saved as a string named CurrentOptions. Since TGR2KML currently only works on area features, the point and line options are disabled in this dialog. (Those options are available in TGR2SHP.)



Options for TIGER 2006 2nd Edition

Controls

Select All | Clear All | ☐ Clip Perennial Water Polygons | ☐ Use these options for all subsequent cases

☐ Roads	☐ AIANHH Current	☐ SDELM Current	☐ ZCTA5 2000
☐ Rails	☐ Alaska Native Regional Corporations 2000	☐ School Districts Secondary 2000	☐ ZCTA5 Current
☐ Misc. Ground Transport	☐ ANRC Current	☐ SDSEC Current	☐ ZCTA3 2000
☐ Landmarks	☐ American Indian Tribal Subdivisions 2000	☐ School Districts Unified 2000	☐ ZCTA3 Current
☐ Physical Features	☐ AITS Current	☐ SDUNI Current	☐ Urban 2000 and Current
☐ Non-Visible	☐ Consolidated City 2000	☐ MSA/CMSA 2000	☐ Traffic Analysis Zones
☐ Hydrography	☐ Consolidated City Current	☐ MSA/CMSA Current	☐ TAZ State Combined
☐ Unknown	☐ County Subdivision 2000	☐ PMSA 2000	☐ Voting Districts
☐ County 2000	☐ County Subdivision Current	☐ PMSA Current	☐ State Upper House 2000 and Current
☐ County Current	☐ Subbarrio 2000	☐ NECMA 2000	☐ State Lower House 2000 and Current
☐ Tract 2000	☐ Subbarrio Current	☐ NECMA Current	☐ Urban Growth Area
☐ Tract Current	☐ Place 2000	☐ 106th and 108th Congressional Districts	☐ Landmark Pts and Polys
☐ Group 2000	☐ Place Current	☐ 110th Congressional Districts	☐ Water Polygons
☐ Group Current	☐ School Districts Elementary 2000	☐ PUMA 5%	☐ Corrected Count Polys
☐ Block 2000		☐ PUMA 1%	☐ All Polys
☐ Block Current			
☐ American Indian/ Alaska Native/ Hawaiian Homeland 2000			

Economic Census Polygons

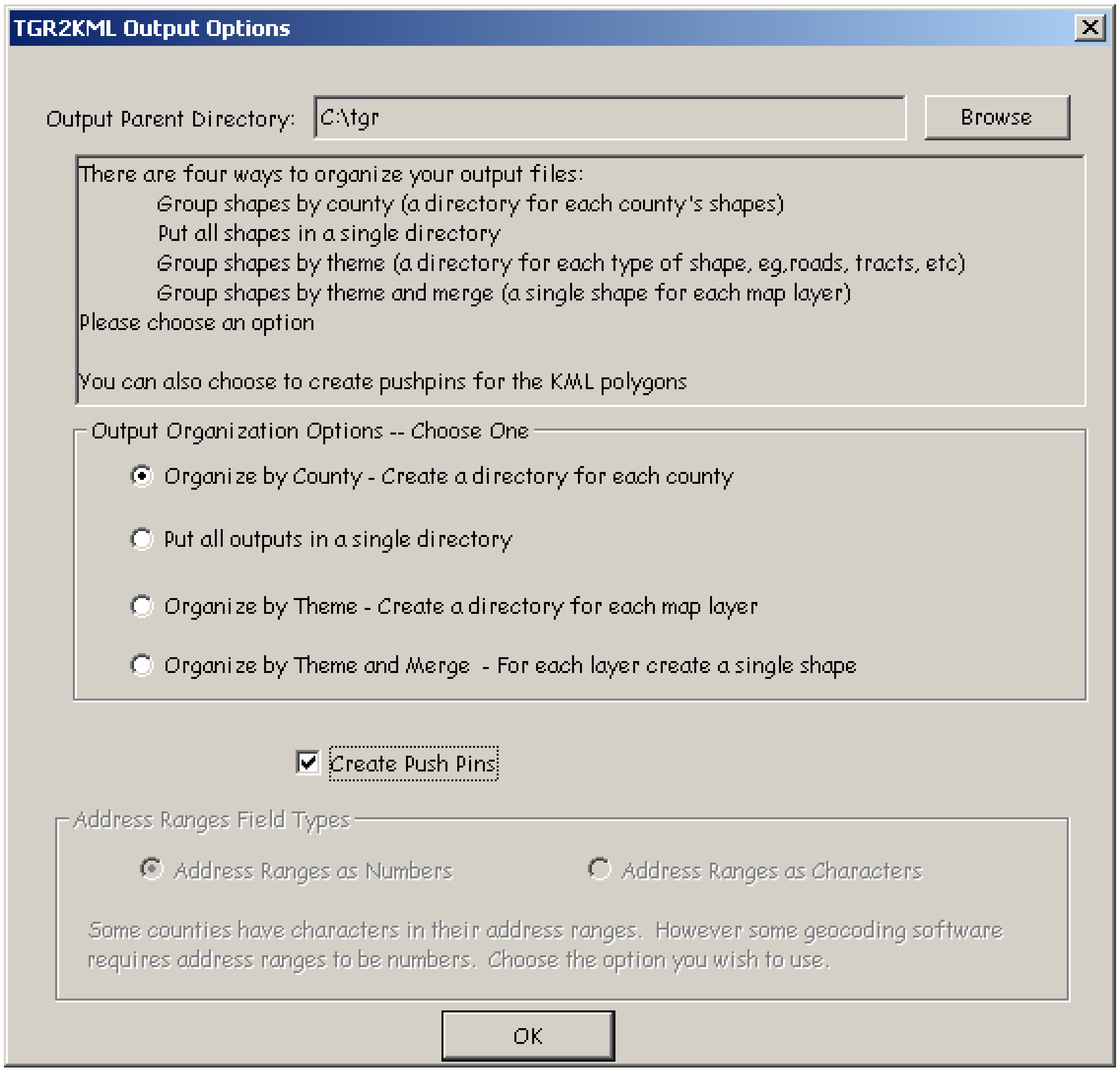
☐ County Economic	☐ Core Based Statistical Areas	☐ New England City + Town Areas
☐ Place Economic	☐ Combined Statistical Areas	☐ Combined NECTAs
☐ Commercial Region Code	☐ Metropolitan Divisions	☐ NECTA Divisions

OK

Figure 8. Layer options dialog. The code for this dialog is in options2.cpp. A similar dialog and file (options.cpp) exists for pre-2002 versions of TIGER

When the user clicks on the GO button in the main dialog, the output options dialog appears (Figure 9). Three values are captured in this dialog:

1. The directory to which the program should write outputs. This is stored in a variable named CurrentDir.
2. How shapes should be organized: by county, in a single directory, by theme, or by theme and merge. The radio button selected is stored in a variable named ChosenOutputOption.
3. Whether push pins should be created. If chosen, the centroid of each area feature is calculated and a push pin is put at that location. This is in addition to the polygons generated. If an area feature is composed of many parts, the centroid of the largest polygon in that feature is used for the push pin.



Output Parent Directory: C:\tgr

There are four ways to organize your output files:
 Group shapes by county (a directory for each county's shapes)
 Put all shapes in a single directory
 Group shapes by theme (a directory for each type of shape, eg,roads, tracts, etc)
 Group shapes by theme and merge (a single shape for each map layer)
Please choose an option

You can also choose to create pushpins for the KML polygons

Output Organization Options -- Choose One

- [✓ Organize by County - Create a directory for each county]
- Put all outputs in a single directory
- Organize by Theme - Create a directory for each map layer
- Organize by Theme and Merge - For each layer create a single shape

[✓ Create Push Pins]

Address Ranges Field Types

[✓ Address Ranges as Numbers] Address Ranges as Characters

Some counties have characters in their address ranges. However some geocoding software requires address ranges to be numbers. Choose the option you wish to use.

Figure 9. The output options dialog. The code of this dialog is in outputoptions.cpp.

The management of these dialogs is straightforward and can be found in the code files referenced in the figure captions. The focus here is on how the values captured in the dialogs are combined to create the call to the proper dll. For the sake of brevity, only the structure of the call to TGRSHP2.DLL is discussed.

```
//this code fragment is part of the OnOK routine in tgrshpdlg.cpp
char *flags = (char *)calloc(86,sizeof(char));
sprintf(flags,"%s",strflags);
if ((flags[0] == '1') && (Adsasnum))
        flags[0] = '2';
char *outdir = (char *)calloc(strlen(strout)+1,sizeof(char));
sprintf(outdir,"%s",strout);
char outopt[] = "0123";
flags[79] = outopt[ChosenOutputOption];
if (Pushpins == 1)
        flags[81] = '1';
else
        flags[81] = '0';
retval = DisplayV2(fips,outdir,flags, i+1, n);
free(flags);

free(outdir);
```

TextBox 1. Creating the call to DisplayV2 in tgrshp2.dll.

A call to the DisplayV2 function of TGRSHP2.DLL takes the following arguments.

- The full name of the input TIGER file, including its path. This name is stored in the variable fips.
- The full path to the output directory, stored in outdir.
- The value of all the flags on the Output Options and the Layer Options dialogs. This is simply a long string of integer values, mostly 0 or 1. Since TGR2KML does not produce line or point files, the flags corresponding to those layers will all be zero.
- The sequence number of the TIGER file being processed.
- The total number of TIGER files to be processed.

Once the call to DisplayV2 is made, control passes to the tgrshp2.dll.

The DLL
The KML files the user requests are generated in the dll. When called, the dll checks input arguments and then calls a routine named ProcessTiger. That controls how each request is processed. That is, it checks what input files need to be read and what layers need to be created. For expository purposes, we will focus on how the program reads the inputs, creates area features, and writes KML files.

Reading TIGER Inputs
As described above, a TIGER “file” is really a set of files organized by what the Census Bureau calls record types. For each record type, tgr2shp creates a data structure corresponding to the fields described in Chapter 6 of the TIGER technical documentation. These are found in the file rtypes.h. For example, the structure for type S records begins as follows (TextBox 2).

```
typedef struct {
        char    RecordType;//1
        int     Version;//4
        int     File; //5
        char    Cenid[6];//5 Census File Identification Code
        char    Polyid[11];//10 Polygon Identification Code
        char    State[3];//2 FIPS State Code, 2000
        char    County[4];//3 FIPS County Code, 2000
        char    Tract[7];//6 Census Tract, 2000
        char    Block[5];//4 Census Block Number, 2000
        char    Blkgrp;//1 Census Block Group, 2000char
```

TextBox 2. Part of the structure for RTS. Compare this TextBox with Figure 5. This is found in the file rtypes.h.

In some structures, fields have been added to aid in processing. For example, the structure for RT1 includes a field MyCfcc which is a single character that keeps track of the line type (road, rail, water, etc). This is useful when processing single line types, such as roads, railroads, and so on.

To process the records for a given record type, three subroutine types are used: one to open the file and allocate the necessary memory space (Get_Type*X*), one to parse each record in the input file (Parse_Type*X*_Record), and one used to sort the resulting records (Compare*X*), where *X* is the record type. In this section, RTS will be used as an example, so the relevant routines are named Get_TypeS, Parse_TypeS_Record, and CompareS. Get_TypeS simply opens the file, scans it to see how many records are present, allocates the memory necessary to hold those records, and then reads each record into that memory. The routine ends by sorting the records.

As each record is read in the Get_Type sub, it is passed to the corresponding parsing function. That function simply chops up the input string into fields of the correct length and type (TextBox 3). Two functions, Nscanf and Ascanf are used to read each field.

```
void Parse_TypeS_Record(long k)
{
        Nscanf(Record,1,"%c",&TypeSs[k].RecordType);
        Nscanf(&Record[1],4,"%d",&TypeSs[k].Version);
        Nscanf(&Record[5],5,"%d",&TypeSs[k].File);
        Ascanf(&Record[10],5,TypeSs[k].Cenid);
        Ascanf(&Record[15],10,TypeSs[k].Polyid);
        Ascanf(&Record[25],2,TypeSs[k].State);
```

TextBox 3. Beginning of the Parse subroutine for RTS.

Once all the records are read, the resulting array is sorted. This is accomplished with a call to qsort with the appropriate comparison function. For example, RTS records are sorted on the concatenated string of CENID and POLYID (TextBox 4).
.

```
int CompareS(RecordTypeS *elem1, RecordTypeS *elem2)
{
        CString el1, el2;
        el1.Format("%s%s",elem1->Cenid,elem1->Polyid);
        el2.Format("%s%s",elem2->Cenid,elem2->Polyid);
        if (el1 > el2)
                return 1;
        if (el1 == el2)
                return 0;
        return -1;
}
```

TextBox 4. The compare function for RTS used in qsort

Processing Polygons

This section describes the logic used in creating polygons. For the sake of brevity, I assume that the necessary record types have been read into the program and sorted as described in the section on Reading Inputs. To illustrate the steps used to build polygons, this section will focus on building Census Tracts as defined in the 2000 Census.

Once the necessary input files are read, processing polygons consists of five major steps:

1. Determining the number of area features of each type in the TIGER archive.
2. Building the relationships between lines and the type of features being processed so that internal lines are removed.
3. Orienting tract boundary lines so that each line-polygon relationship is based on a “right-hand-side” orientation.
4. Organizing the tract boundary lines for efficient processing of each polygon.
5. Assembling the lines that define a specific type of polygon, in this case Census Tracts 2000, into rings.

Each of these steps is described below.

Step 1. Determining the Number of Area Features of Each Type in the TIGER Archive.

The relationship between lines and type S polygons, such as Census Tracts 2000, is created in the subroutine Build_TypeI_BordersS. After some initialization, a one dimensional array of polygon counts, SPolycounts is set to zero. Each element in that array corresponds to a type of polygon referenced in RTS (TextBox 5)

```
void Build_TypeI_BordersS()
{
        int num, steppos, i;
        int indexl, indexr;
        CString info;
        CString leftval, rightval;

        for (i = 0; i < 31; i++)
                SPolycounts[i] = 0;
```

TextBox 5. Initializing variables and arrays

A data structure, RecordTypeIS, is used to hold the S polygon left and right values for each line and polygon type. This structure definition can be found in the file rtypes.h. Here a portion of that structure is repeated.

```
typedef struct{
        char    Countyl[6];
        char    Tractl[12];
        //For the sake of brevity, I have left out a large number of declarations
        char    Countyr[6];
        char    Tractr[12];
}RecordTypeIS;
```

TextBox 6 The RecordTypeIS structure

One instance of this structure, called TypeISs, is created. In Build_TypeI_BordersS that instance is initialized so that each left and right value is assigned a blank string, or in the case of water flags, a value of 0. The text below contains some of that code.

```
sprintf(TypeISs.Countyl,"     ");
sprintf(TypeISs.Tractl,"           ");
//For the sake of brevity, I have left out a large number of initialization steps
sprintf(TypeISs.Countyr,"     ");
sprintf(TypeISs.Tractr,"           ");
```

TextBox 7. Initializing the TypeISs structure

For each line in Record Type I, i.e., for each line in TIGER, there usually will be a basic census polygon to the left and to the right. However, if the line is on the external boundary of a county, then either the polygon on the left or on the right of that line will be blank. A search routine, called Get_SIndex, is used to find corresponding line in the RTS records. Since the records in RTS are sorted based on the CENID+POLYID value (see the section on Reading Inputs, above), finding the TypeS record for a particular

CENID, POLYID pair is quite quick. The value returned is used to track the TypeS record that corresponds to either the left or the right of the line (TextBox 8).

```
TypeISIndex[num].indexl = indexl = Get_SIndex(TypeIs[num].Cenidl, TypeIs[num].Polyidl);

TypeISIndex[num].indexr = indexr = Get_SIndex(TypeIs[num].Cenidr, TypeIs[num].Polyidr);
```

TextBox 8 Getting the index numbers for the lines

If the value returned by Get_SIndex is non-negative, then the polygon corresponding to that index (either left or right) is "meaningful." That is, it is not outside the TIGER archive being processed. If that is the case, then the values in the TypeIS data structure are given their proper values from RTS. The TextBox 9 below contains some of that code. In this code, the state and county FIPS codes are written to the TypeISs County. Similar calculations are made for tracts, block groups, blocks, places, and so on.

```
//this is only a code fragment.  See the source code for the entire setting of left/right areas
if (indexl >= 0)
    {     //we have a polygon border
    sprintf(TypeISs.Countyl,"%2s%3s",TypeSs[indexl].State,TypeSs[indexl].County);
}
if (indexr >= 0)
    {     //we have a polygon border
    sprintf(TypeISs.Countyr,"%2s%3s",TypeSs[indexr].State,TypeSs[indexr].County);
}
```

TextBox 9. Updating the county values on the left and right of the line

Once the values of TypeISs on the left and right for the current line are determined, they can be examined to see if the line is an external boundary for each type of polygon described in RTS. Consider, for example, tracts. If the tract on the left of a line and the tract on its right are the same, then the line does not define a tract boundary. (Note: Tract IDs must be fully articulated. That is, they must composed of the state FIPS, the county FIPS, and the tract FIPS.) If they are different, then the line does define a tract boundary. Further, if they are different and neither one is blank then the line defines a boundary between two tracts in that TIGER file. If one of them is blank, then one side of the line is not part of the current TIGER file (i.e., it belongs to another county). The code in TextBox 10 illustrates this logic. There is one if block like that in TextBox 10 for each record type S polygon.

```
if (strcmp(TypeISs.Tractl,TypeISs.Tractr) != 0)
{
            SPolycounts[2]++;
            if ((strcmp(TypeISs.Tractl,"          ") != 0) &&
                        (strcmp(TypeISs.Tractr,"          ") != 0))
            SPolycounts[2]++;
}
```

TextBox 10. Determining the number of lines that define tract boundaries

While the above example refers to type S polygons (those defined for Census 2000), there are similar "Build_TypeI_Borders" subroutines for record types A (current geography polygons), B (correction count polygons), E (economic census polygons) and 8 (landmark polygons).

Step 2. Building the Relationships Between the Lines and the Type of Feature Being Processed and Removing Internal Lines

Once the line information in record type I is related to specific types of census polygons, those polygons can be assembled. This is accomplished in subroutines named "Match_Poly_Lines_*polytype*", where polytype refers to a specific polygon type. In the example below, the code from Match_Poly_Lines_Tracts00 is described. Tracts00 refers to Census Tracts for Census 2000.

The routine begins by defining some variables (TextBox 11).

```
void Match_Poly_Lines_Tract00()
{
            int polyid, found, i, nparts, ptcount;
            long num, n, written, curcase;
            Tract_Border_Type *tract;
            double  curX, curY;
            FILE  *dbftmp, *ptstemp, *headtemp, *partstemp, *boxtemp, *shxtemp;
            CString fname, info, tname;
            char *tvalue;
            char trtrecord[92]; //size of tract dbrecord + 2
            int steppos;
            CString errmsg;
            int doleft, doright;
            int bval;
            CString repval;
            int curpolys;
```

TextBox 11. Initializing the Match_Poly_Lines variables.

The meanings of most of these variables are straightforward. However, two deserve more attention. The first is the pointer to the Tract_Border_Type. This structure contains information necessary for storing borders of the tract polygons. Defined in the file rtypes.h, the structure stores a tract ID number (the STFID), a line ID, the start and end coordinates of the line, and two integer values, Flip and Chosen. A key step to assembling the lines that define a polygon, tracts in this example, is to orient all lines so that the tract being processed is always on the right side of line. This is what Flip indicates. A value of 0 indicates that the line was digitized so that the tract being processed was on the right, while a value of 1 indicates that the line had to be reversed so that the tract was always on the right. Orienting the lines in this manner aids in constructing complete rings, and determining which rings define holes. It also allows the program to distinguish between an outer boundary, which will be clockwise, and an inner boundary (hole) which will be counterclockwise.   The meaning of Chosen will become clear below. (Note: it would be more correct to define Flip and Chosen as Booleans.) Every type area feature, eg, counties, tracts, places and so on, has a Border_Type structure.

The second variable worth considering is string trtrecord. Each feature has an attribute record. The contents of those records are defined in the file Dbase.h. TextBox 12 presents the field definitions for tract records.

```
FieldData dbTract[] = {
        {"GIST_ID",'N',8,0,0},
        {"FIPSSTCO",'C',5,0,0},
        {"TRACT",'C',6,0,0},
        {"STFID",'C',11,0,0},
        {"NAME",'C',60,0,0},
        {0,0,0,0,0}
};
```

TextBox 12. The field definitions for the tract attribute file.

The FieldData structure contains an entry for each field. The entries consist of the field name, its type, its length, the number of decimal points, and a flag indicating if the field has been indexed. (For TGR2KML, this flag is always set to “0”.) The string trtrecord is sized to the sum of the lengths of the field data + 2. The two extra units are for the string terminator and a leading blank on each record. Thus, for tracts, the value is 92 or 8+5+6+11+60+2.

The next section of the code checks to make sure there are cases to process (did the Build_TypeI_BordersS routine find any cases?), writes some information to the screen about the current process, and then opens some temporary files to store information needed to construct the KML file. (Note: The information needed to construct shapes in TGR2SHP is also calculated here. For example, the bounding box for each area feature is determined. These values are not necessary for KML files, but were left in by the author.) Writing KML files is discussed in more detail in a later section. For now, let’s continue on to the processing of the files.

The program grabs enough memory to hold all the polygon border information it will need. This value is based on the SPolycounts value for Tracts2000, and the number of water polygons that might be included (TextBox 13).

```
tract = (Tract_Border_Type *)calloc(SPolycounts[2]+Watercut*SPolycounts[0],
            sizeof(Tract_Border_Type));
```

TextBox 13. Allocating memory to hold the tract border lines

After some values are initialized, a loop is used to check every line to see if it defines the boundary of the feature being processed, in this case Tracts 2000. The TypeISs values needed to make that determination are retrieved and examined to see if the line defines a tract boundary (TextBox 14).

```
if ((strcmp(TypeISs.Tractl,TypeISs.Tractr) == 0) &&
   (strcmp(TypeISs.Countyl,TypeISs.Countyr) == 0))
{
            if (Watercut)
            {
                        if (TypeISs.Waterl == TypeISs.Waterr)
                                    continue;
                        if (abs(TypeISs.Waterl - TypeISs.Waterr) > 1)
                                    continue;
            }
else
            continue;
}
```

TextBox 14. Determining if a line is a tract boundary

For expository purposes, let us assume that water polygons are not being cut from shapes. In that case, the if statements on Waterl and Waterr do not come into play. The first if statement checks to see if the county and tract on the left equal the county and tract on the right. If this is true, then the line does not define a tract boundary. If, instead, the program does not hit a continue statement, then the line does define a tract boundary.

In Figure 10, the lines that are boundary lines are solid, the internal lines are dashed, and the from and to nodes for each line are red dots. This is a typical case before the if statements in TextBox 14 are executed. After they are executed, the remaining lines are as in Figure 11.

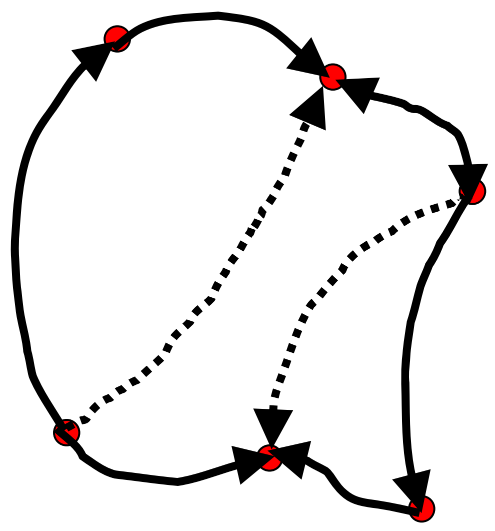


Figure 10. A hypothetical tract before the if statements in TextBox 18 are executed

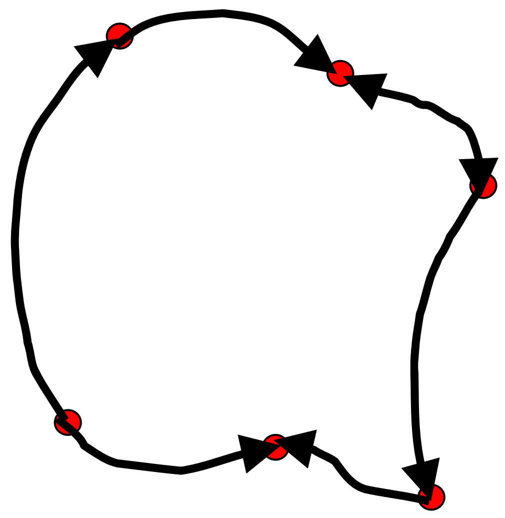


Figure 11. The same tract after the internal lines are removed

Step 3. Orienting the Boundary Lines

The code that immediately follows that in TextBox 14 gathers all the information about the tract to the left and the right of each boundary line. Consider the left side of the line (TextBox 15). The opening if statement makes sure there is a tract on the left side of the line. If so, that tract's full FIPSCODE and the line's TLID are written to the Tract_Border_Type. This is followed by specifying the start and end coordinates of the line. Finally, since the tract is on the left of the line, the value of Flip is set to 1. This will cause the line to be reoriented so that the tract is on its right.

```
doleft = 0;
if ((strcmp(TypeISs.Tractl,"      ") != 0) && (sindexl >= 0))
{//left case
        if (Watercut)
        {
                if(TypeISs.Waterl != 1)
                        doleft = 1;
                }
        else
                doleft = 1;
        if (doleft)
        {
                sprintf(tract[written].Id,"%s%s",TypeISs.Countyl,TypeISs.Tractl);
                sprintf(tract[written].Tlid,"%s",TypeIs[num].Tlid);
                tract[written].Frlong = Type1s[num].Frlong;
                tract[written].Frlat = Type1s[num].Frlat;
                tract[written].Tolong = Type1s[num].Tolong;
                tract[written].Tolat = Type1s[num].Tolat;
                tract[written].Chosen = 0;
                tract[written].Flip = 1;
                written++;
        }
}
//similar code exists for the right side polygon, but Flip is 0
```

TextBox 15. Populating the Tract_Border_Type structure for a tract that is on the left of a line

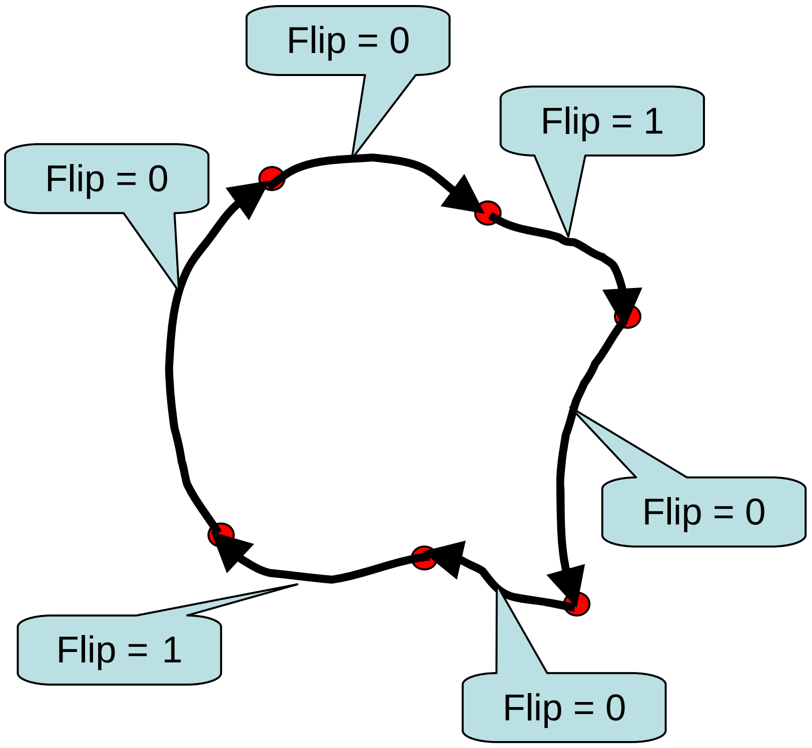


Figure 12. The tract border lines after Flip is set

A similar section of code does the same thing for the polygons on the right of the line. However, in that case the value of Flip is set to 0. The results of this step are depicted in Figure 12.

Once all the lines have been processed, the program checks to make sure that no more Tract_Border_Type lines have been written than memory that has been allocated. It then writes some information to the screen and checks that at least one element of type Tract_Border_Type has been found. If not, it closes open files and the subroutine is exited.

Step 4. Organizing the tract boundary lines for efficient processing of each polygon

The next step is crucial if the polygons are to be assembled efficiently. The lines that have been written to the Tract_Border_Type structure are sorted (TextBox 16).

```
qsort((void *)tract,written,sizeof(Tract_Border_Type),(compfn)CompareTract);
```

TextBox 16. Sorting the Tract_Border_Type elements

The comparison function used in this sort is given in TextBox 17.

```
int CompareTract(Tract_Border_Type * elem1, Tract_Border_Type * elem2)
{
        return(strcmp(elem1->Id,elem2->Id));
}
```

Consider what the memory pointed to by tract looks like before and after sorting. Before the sort the elements of tract are sorted by TLID. Table 1 presents a simplified version of that data. In this table we can see that line 1 is a boundary between two Census Tracts (1 and 3), with 1 on its left and 3 on its right. Line 2 is an external boundary. That is, it defines a tract on its right, but on its left is another county. We know this because there is no entry for line 2 with a flip value of 1. However, lines that define a specific tract are difficult to find. For example, finding the lines that define tract 1 would require looking through the entire “Before qsort” table.

Before qsort

TractID	LineID	Start	End	Flip	Chosen
1	1	X,Y	X,Y	1	0
3	1	X,Y	X,Y	0	0
1	2	X,Y	X,Y	0	0
2	3	X,Y	X,Y	0	0
3	3	X,Y	X,Y	1	0
3	4	X,Y	X,Y	1	0
2	4	X,Y	X,Y	0	0
1	5	X,Y	X,Y	1	0
4	6	X,Y	X,Y	0	0
1	6	X,Y	X,Y	1	0

Table 1. The Tract_Border_Type elements before sorting

Performing the sort partially address this problem (Table 2). The sorting puts all the lines that define a particular tract into blocks of contiguous memory. This means that rather than searching all the lines that define tract boundaries in order to find the next line in a chain, we only need search a greatly reduced subset of lines. For example, to find the lines that define tract 3, we only need to look at the seventh through ninth entries. (In the table these records are in bold-italics.) In large files like TIGER where counties can have over 100,000 lines, this greatly speeds up processing.

After qsort

TractID	LineID	Start	End	Flip	Chosen
1	1	X,Y	X,Y	1	0
1	2	X,Y	X,Y	0	0
1	5	X,Y	X,Y	1	0
1	6	X,Y	X,Y	1	0
2	3	X,Y	X,Y	0	0
2	4	X,Y	X,Y	0	0
3	***1***	***X,Y***	***X,Y***	***0***	***0***
3	***3***	***X,Y***	***X,Y***	***1***	***0***
3	***4***	***X,Y***	***X,Y***	***1***	***0***
4	6	X,Y	X,Y	0	0

Table 2. The Tract_Border_Type structure after the sort

Even though all the lines that define a tract are in a contiguous block of memory there is still work to be done. For tract 3, for example, we still have to determine if line 1 connects to line 3 or line 4, and so on until we complete an entire ring.

The previous steps have built the relationship between lines and features by exploiting the topological relationships between record types I and S. This allowed us to assemble all the lines that define area features of a particular type, in this case tracts, and to orient them in the proper direction. The sorting of that result groups all the lines that define each feature into contiguous blocks of memory. We can now proceed to step 5.

Step 5. Assembling Lines into Rings

Before looking at the sections of code that put the lines in the correct order for each feature, it is necessary to consider the difference between features and rings. A feature can consist of multiple rings. Figure 13 illustrates these concepts. Each case represents a single area feature. In Case A, the feature consists of a single ring. This is often the case. However, there may be cases like those in Case B. In this case the feature consists of two rings. For example, two islands may form a tract. In Case C, there is only one “part” to the feature, but there is a hole in it. Thus, finding a series of lines that forms a closed loop for a given feature does not guarantee that processing that feature is complete. It only means that a ring has been found. Further checks must be made to see if there are other rings that help define the feature. Fortunately, the sorting used in the previous step makes this straightforward.

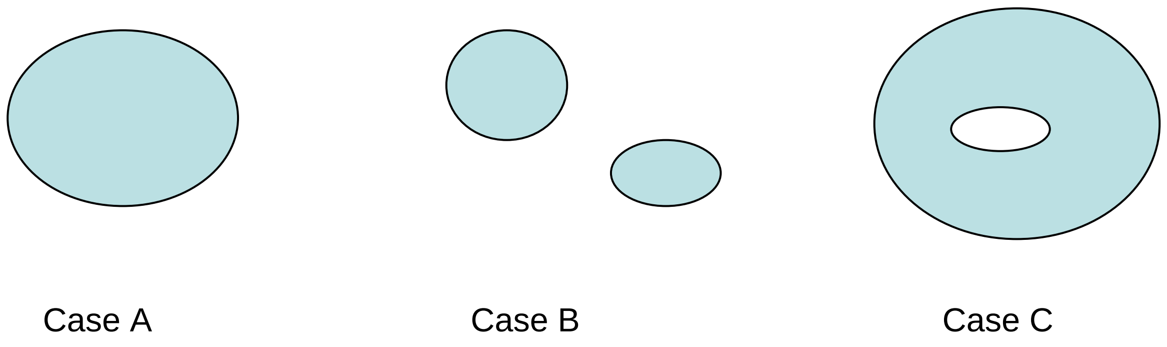


Figure 13. Shapes and Rings

Starting from the first record in this structure, several variables are initialized (TextBox 17).

```
FileMinX = FileMinY = LARGE;
FileMaxX = FileMaxY = -LARGE;
CurrentMinX = CurrentMinY = LARGE;
CurrentMaxX = CurrentMaxY = -LARGE;
curpolys = CurNumElements;
curcase = 0;
polyid = 1;
Npts = 0;
nparts = 1;
ptcount = 0;
```

TextBox 17. Initialization of variables needed to assemble area features

The meaning of these is as follows:

- The File and Current min and max values are used to store the bounding box of the entire shape file and the current feature, respectively. These are holdovers from TGR2SHP.
- CurNumElements is the number of features written to the output shape before the current county was processed. In most cases this will be zero. However, if the output option is to merge several TIGER files (e.g., tracts from several counties), then CurNumElements is the count of tracts for all counties processed previous to the current county.
- polyid is the id of the current feature. As each feature is processed, polyid is incremented by one.
- Npts is the number of points in the current line
- Nparts tracts the number of rings in a feature
- Ptcount is number of points in the current ring

At the start of each feature, its attribute values are written to the string dimensioned to hold its database record. In the current example, this would be trtrecord. Once populated that record is written to a temporary holding file (TextBox 18).

```
tract[curcase].Id[11] = '\0';
sprintf(trtrecord," %8d",polyid+curpolys);//changed
sprintf(&trtrecord[9],"%-5s",tract[curcase].Id);
sprintf(&trtrecord[14],"%-6s",&tract[curcase].Id[5]);
sprintf(&trtrecord[20],"%-11s",tract[curcase].Id);
tvalue = (char *)calloc(7,sizeof(char));
sprintf(tvalue,"%-6s",&tract[curcase].Id[5]);
tvalue[6] = '\0';
tname = Get_TypeC_Name2(11,tvalue,"",2000);
sprintf(&trtrecord[31],"%-60s",tname);
free(tvalue);
fwrite(&trtrecord,sizeof(trtrecord)-1,1,dbftmp);
```

TextBox 18. Writing the current shape's attributes

The next section of code checks if the line is flipped, writes the beginning coordinates, gets the intermediate points stored in record type 2, and then writes the ending points. Notice that if the line is flipped, the coordinates are written in the opposite direction of the way they were stored in TIGER. Once all the points for a line are assembled, ptcount is set to its new value and the points are written to a temporary file (TextBox 19).

```
if (tract[0].Flip == 0)
{
        curX = tract[0].Tolong;
        curY = tract[0].Tolat;
        Pts[Npts].Longitude = tract[curcase].Frlong;
        Pts[Npts].Latitude = tract[curcase].Frlat;
        Npts++;
        Get_Shape_Points(tract[curcase].Tlid, tract[curcase].Flip);
        Pts[Npts].Longitude = tract[curcase].Tolong;
        Pts[Npts].Latitude = tract[curcase].Tolat;
        Npts++;
        tract[curcase].Chosen = 1;
}
else
{
        curX = tract[0].Frlong;
        curY = tract[0].Frlat;
        Pts[Npts].Longitude = tract[curcase].Tolong;
        Pts[Npts].Latitude = tract[curcase].Tolat;
        Npts++;
        Get_Shape_Points(tract[curcase].Tlid, tract[curcase].Flip);
        Pts[Npts].Longitude = tract[curcase].Frlong;
        Pts[Npts].Latitude = tract[curcase].Frlat;
        Npts++;
        tract[curcase].Chosen = 1;
}
ptcount += Npts;
fwrite(Pts,sizeof(Geographic_Coordinate),Npts,ptstemp);
```

TextBox 19. Gathering the line points and writing them in the proper order

Get_Shape_Points gets all the intermediate points in a line from RT2 and puts them in the proper order, which is dependent on the value of Flip. While gathering each point, the bounding box limits for the file and the current feature are checked. The next bit of code in Match_Poly_Lines_Tract00 checks if the start and ending points of the line affect these bounding boxes. That code is not repeated here, but can be found in the source code.

Once the first line of the shape file is processed, a loop is constructed to cycle through all the remaining items in the Tract_Border_Type data structure. Two variables are initialized to 1: n and bval. The first variable, n, keeps track of the number of elements of tract that have been found and processed. This value can never exceed the number of Border_Type_Records found in step 2. The second value, bval, is used to track the position of the first record in tract that has the current tract ID. Looking at Table 3, one can see the following. If the tract being processed has ID 1, then bval will point to first record. If the tract ID is 2, bval will point to the fifth record. If the tract ID is three, then the bval will point to the seventh record, and so on.

		After qsort			
TractID	**LineID**	**Start**	**End**	**Flip**	**Chosen**
1	1	X,Y	X,Y	1	1
1	2	X,Y	X,Y	0	0
1	5	X,Y	X,Y	1	0
1	6	X,Y	X,Y	1	0
2	3	X,Y	X,Y	0	0
2	4	X,Y	X,Y	0	0
3	1	X,Y	X,Y	0	0
3	3	X,Y	X,Y	1	0
3	4	X,Y	X,Y	1	0
4	6	X,Y	X,Y	0	0

Table 3. The sorted Tract_Border_Type structure

The key to building area features quickly is to only search that section of the tract memory block that pertains to that feature. Suppose, for example, that tract 1 is composed of the following set of line segments: 1, 5, 6, and 2. At the start of the while loop, a Boolean, found, is set to 0 and a check is made to see if bval points to a record that has a TractID that is higher (later in the sort) than the current case. If it is, then bval is reduced until it (bval) is pointing to a record in with the proper tract ID (TextBox 20).

```
while (n < written)
{
        found = 0;
        while (bval > 1)
        {
                if (strcmp(tract[bval].Id, tract[curcase].Id) >= 0)
                        bval--;
                else
                        break;
        }
```

TextBox 20. Positioning bval to point to the first value with the ID of the current case

Once bval is pointing to a record with the correct tract ID, a for loop over a counter i is used to look at each record. Although this loop is set to run from bval to the total number of records, only those records that pertain to the current tract ID will be processed. This is accomplished through a series of if tests.

Test 1: Is the value of Chosen for the ith line = 0? If not, move to the next line. Otherwise perform the next test.
Test 2: If Test 1 is passed, is the value of the tract ID for the ith line greater than the current tract ID? If so, then all the lines for the current tract ID have been processed. Set bval = i and break out of the for loop. Otherwise, perform the next test.
Test 3: If Tests 1 and 2 are passed, then check if the ith line has the same tract ID as the current case. If not, get the next line.

If the third test is passed, then the ith line must have the correct tract ID. The line is then checked to see if it connects to the curcase line. If it does, its points are gathered, its value of Chosen is set to 1, as is the value of found. Here is the code for the loop (TextBox 21).

```
for (i = bval; i < written; i++){
        Npts = 0;
        if (tract[i].Chosen != 0)
                continue;
        if (strcmp(tract[i].Id,tract[curcase].Id) > 0)
        {
                bval = i;
                break;
        }
        if (strcmp(tract[i].Id,tract[curcase].Id) != 0)
                continue;
        if (tract[i].Flip == 0)
        {
                if ((tract[i].Frlong == curX) && (tract[i].Frlat == curY))
                {
                        Get_Shape_Points(tract[i].Tlid,tract[i].Flip);
                        Pts[Npts].Longitude = tract[i].Tolong;
                        Pts[Npts].Latitude = tract[i].Tolat;
                        Npts++;
                        n++;
                        steppos = (int)(n*100/written);
                        tract[i].Chosen = 1;
                        found = 1;
                        curX = tract[i].Tolong;
                        curY = tract[i].Tolat;
                        curcase = i;
                        break;
                }
        }
        else
        {
                if ((tract[i].Tolong == curX) && (tract[i].Tolat == curY))
                {
                        Get_Shape_Points(tract[i].Tlid,tract[i].Flip);
                        Pts[Npts].Longitude = tract[i].Frlong;
                        Pts[Npts].Latitude = tract[i].Frlat;
                        Npts++;
                        n++;
                        steppos = (int)(n*100/written);
                        Ptrdlg->m_Progress.SetPos(steppos);
                        curX = tract[i].Frlong;
                        curY = tract[i].Frlat;
                        tract[i].Chosen = 1;
                        found = 1;
                        curcase = i;
                        break;
                }
        }
 }
```

TextBox 21 Getting the next line for a tract

Once the first for loop within the while loop is processed the lines that define a closed loop for the current tractID have been collected. Recalling Figure 13, it is clear that the loop might be the entire tract, or it might be ring in the tract.

A second for loop is executed to see if the loop completes the tract or if it is just a ring (TextBox 22). Here is the logic. If a complete loop of lines is constructed, found will be zero. If that is so, we check to see if, starting at bval, any line with the current tract ID has a value of Chosen = 0. This would indicate that the line has yet to be processed. Therefore, the current closed loop must be a ring, not a complete feature.

```
if (found == 0) //initialize new case, ie, new ring or new feature
{
        fwrite(&ptcount,sizeof(int),1,headtemp);
        ptcount = 0;
        while (bval > 1)
        {
                if (strcmp(tract[bval].Id, tract[curcase].Id) >= 0)
                        bval--;
                else
                        break;
        }
        for (i = bval; i < written; i++) //check for new ring in feature
        {
                if (tract[i].Chosen != 0)
                        continue;
                if (strcmp(tract[i].Id, tract[curcase].Id) > 0)
                        break;
                if (strcmp(tract[i].Id, tract[curcase].Id) != 0)
                        continue;
                //if we get here it is a new ring in current feature
                found = 1;
```

TextBox 22 Checking if the closed loop is a ring or a complete feature

The remainder of the for loop writes out the first line segment of the next ring. It’s code is similar to that used above (checking the value of Flip, calling Get_Shape_Points, and so on) and is not repeated here.

In the above for loop, if the current closed polygon is a ring then found is set equal to 1. If it is not a ring, i.e., if the next record with Chosen = 0 is for a new feature, then the loop is exited with found equal to 0. Thus, the loop for initiating a new feature is only executed if a new feature needs to be started (TextBox 23).

```
if (found == 0) // new feature, first ring
{
        fwrite(&nparts,sizeof(int),1,partstemp);
        nparts = 1;
        fwrite(&CurrentMinX,sizeof(double),1,boxtemp);
        fwrite(&CurrentMinY,sizeof(double),1,boxtemp);
        fwrite(&CurrentMaxX,sizeof(double),1,boxtemp);
        fwrite(&CurrentMaxY,sizeof(double),1,boxtemp);
        CurrentMinX = CurrentMinY = LARGE;
        CurrentMaxX = CurrentMaxY = -LARGE;
        while (bval > 1)
        {
                if (strcmp(tract[bval].Id, tract[curcase].Id) >= 0)
                        bval--;
                else
                        break;
        }
        for (i = bval; i < written; i++)//first check for new ring in current poly
        {
                if (tract[i].Chosen != 0)
                        continue;
                polyid++;
                tract[i].Id[11] = '\0';
                sprintf(trtrecord," %8d",polyid+curpolys);//changed
                sprintf(&trtrecord[9],"%-5s",tract[i].Id);
                sprintf(&trtrecord[14],"%-6s",&tract[i].Id[5]);
                sprintf(&trtrecord[20],"%-11s",tract[i].Id);
                tvalue = (char *)calloc(7,sizeof(char));
                sprintf(tvalue,"%-6s",&tract[i].Id[5]);
                tvalue[6] = '\0';
                tname = Get_TypeC_Name2(11,tvalue,"",2000);
                sprintf(&trtrecord[31],"%-60s",tname);
                free(tvalue);
                fwrite(&trtrecord,sizeof(trtrecord)-1,1,dbftmp);
//the remainder of this section is similar to that used earlier
//i.e., checking the value of Flip, calling Get_Shape_Points, and so on
```

TextBox 23. The code for initializing a new feature

This section starts by checking if found is 0. If so, then the next record in tract is for a new feature (that is, a tract with a different tractID). Before processing the new tract, the bounding box for the current tract is written to a temporary file. Next, bval is set to the last tract record with the ID of the current case (curcase). Starting from that point, the next line yet to be processed is found. For that line, the attribute record for the feature is calculated and written to the trtrecord structure, which is then written to a file. The remainder of the code in this section gets the points for this line, curcase is updated to i, and the value is Chosen is updated. Since this is similar to the code used for the initial line, the code is not repeated here.

The last part of the while loop (TextBox 24) writes out all the points collected so far and checks if the bounding boxes for the current feature and file need to be updated for the first and last points in the current line.

The remainder of the Match_Poly_Line_Tract00 subroutine writes out the bounding box, point count, and parts count for the final feature. In addition, any temporary pointers are freed and a call to a routine that combines the information in the temporary files into a KML file, Build_Poly_KML, is made. That routine is covered in the section on Writing KMLs.

```
fwrite(&nparts,sizeof(int),1,partstemp);
fwrite(&CurrentMinX,sizeof(double),1,boxtemp);
fwrite(&CurrentMinY,sizeof(double),1,boxtemp);
fwrite(&CurrentMaxX,sizeof(double),1,boxtemp);
fwrite(&CurrentMaxY,sizeof(double),1,boxtemp);
fclose(boxtemp);
fclose(headtemp);
fclose(ptstemp);
fclose(partstemp);
fclose(dbftmp);
free(tract);
Build_Poly_KML("trt00",dbTract,sizeof(trtrecord) – 1,"FIPSSTCO");
```

TextBox 24. Completing the Match_Poly_Lines_Tract00 subroutine

Writing KMLs

Before considering the code for writing the KML file, it is necessary to understand how polygons are stored in KML. In KML, a polygon consists of at least one ring that defines an outer boundary. If there are holes in the polygon, as in Figure 13, Case C, then the polygon will have one outer boundary and one or more inner boundaries, with each inner boundary defining a hole. Note that a polygon in KML can have one and only one outer boundary. The question arises as to how to best handle multipart area features, as in Figure 13, Case B. In TGR2KML, multipart area features are put in a KML folder. All the parts of a multipart feature are placed into a folder. Figure 14 illustrates this. Loudon County, Tennessee is a single county that is comprised of four distinct polygons. There is a folder in the Table of Contents (TOC) named 47105 (the FIPSCODE for Loudon County). In that folder there are four distinct polygons, one for each polygon that makes up this feature. I have added red circles to the figure in order to highlight the smaller polygons of this county feature. Note that the folder approach allows the user to turn each part of a multipart feature off or on, something that is not easy with other formats such as shape files. Also note that the two other county entries in the TOC are for counties that consist of only one polygon each (47001 and 47903).

We can now consider the code to generate those outputs. The call to Build_Poly_KML takes four arguments. Here is an example call:

```
Build_Poly_KML("trt00",dbTract,sizeof(trtrecord) – 1,"FIPSSTCO");
```

The meaning of each argument is as follows:

- The file type ("trt00"), which is used to name the output file.

- The field data type (dbTract).
- The size of an attribute record (sizeof(trtrecord) – 1).
- The field which contains the value to be used to name each area feature (“FIPSSTCO”). These are the values that appear in the table of contents in Google Earth. (See Figure 14.)

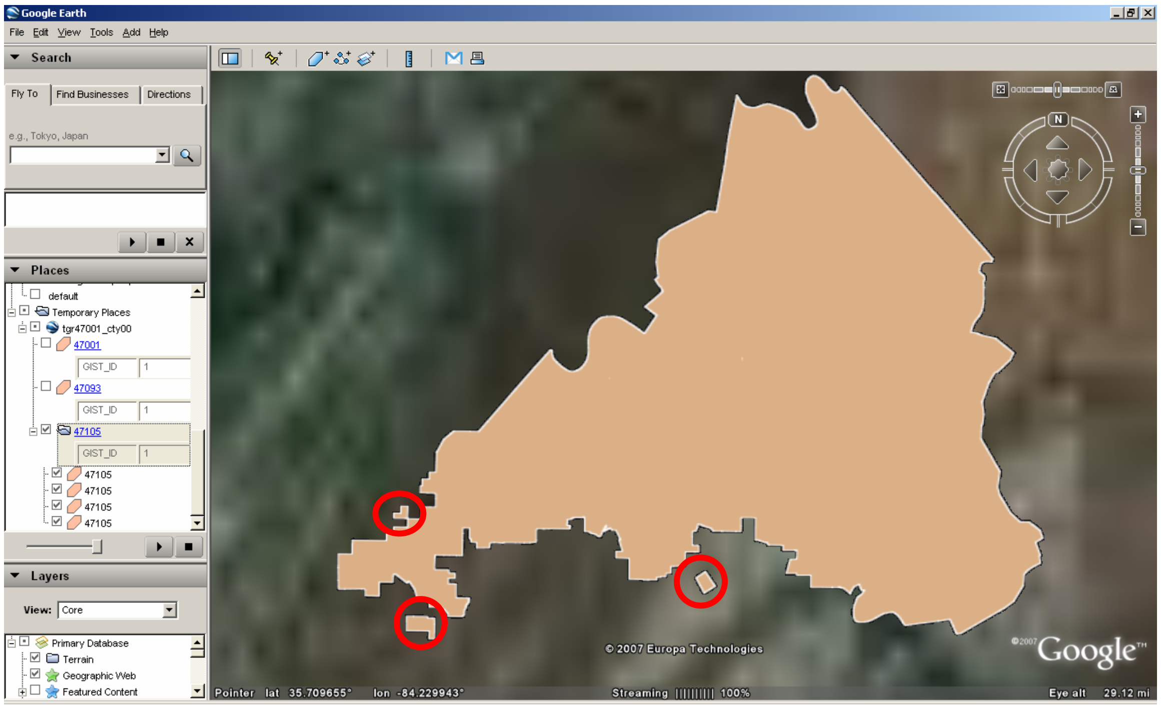


Figure 14. A Multipart Area Feature Consisting of Four Polygons

The Build_Poly_KML subroutine starts by declaring certain variables and structures. Most of these are straightforward. However, the Polyatt structure (TextBox 25) deserves more attention. This structure is used when processing rings. As each closed loop of vertices, or a ring, is processed the program will determine if it defines an inner ring (hole) or an outer ring. If it is a hole, the outer ring in which it is inside is stored in the value Containedin;. The Area of each ring feature, its centroid, and its points are also stored in this feature. The details of how these values are found and manipulated are given below.

```
typedef struct{
            double  Area;
            double  X;
            double  Y;
            point_t* Curpoints;
            BOOLEAN Ishole;
            int Containedin;
            int Ptcount;
}Polyatt;
```

TextBox 25 The Polyatt Structure

Once the necessary variables and structures are declared, the program next performs some housekeeping steps, such as opening the necessary temp files and the proper output file. The most difficult of the four output options (see text associated with Figure 9 for a discussion of the output options) is merging. That is, adding features to a KML file that may already have elements from other counties in it. In the merge option is being used then it is necessary to scan the KML file, if it exists, to find the location where new features should be written. The code for doing this is in TextBox 26. In this code, each line of an existing KML file is scanned until the string "</Document>" is encountered. The beginning of that line is where new features are to be added.

```
if (OutOption == '3')
{
    out = fopen(fname,"r+");
    if (out == NULL)
    {
        out = fopen(fname,"w");
    }
    else
    {
        found = false;
        int linenum = 0;
        //read file to /Document line
        while (!found)
        {
            if (fgets(buffer, 1000, out) != NULL)
            {
                Position = ftell(out);
                if (strstr(buffer,"</Document>") != NULL) //found position to start writing new
                {
                    found = true;
                    fseek(out,(long)oldposition,SEEK_SET);
                    break;
                }
                oldposition = Position;
                linenum++;
            }
            else
            {
                AfxMessageBox("file error");
                return;
            }
        }
    }

}
else
    out = fopen(fname, "w");
```

TextBox 26. Finding the position in which to start writing

The next section of code opens the relevant temp files generated in the Match_Poly_Lines_*polytype* routine. That code is straightforward and not repeated here.

In order to put a name with each feature in the TOC (such as 47105 in Figure 14), the position of the name field in each record of the attribute temp file has to be determined. The attribute temp file is a binary file created in the Match_Poly_Lines_*polytype* sub. The relevant code for determining that position is in TextBox 27.

```
//get position of name field
m = 0;
idposition = 1;
namelen = 0;
while (polytype[m].name != NULL)
{
        if (strcmp(polytype[m].name, namefield) != 0)
                idposition += polytype[m].len_field;
        else
        {
                namelen = polytype[m].len_field;
                featname = (char *)calloc(namelen,sizeof(char));
                break;
        }
        m++;
}
```

TextBox 27. Finding the location of the name field

If the KML polygons are to be put into a new file, a call is made to Build_Kml_Header, a routine that writes out the file header (TextBox 28). In the current version of TGR2KML, the color code for a polygon is hard coded to beige.

```
void Build_Kml_Header(FILE *out, char *type)
{
    fprintf(out,"<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n");
    fprintf(out,"<kml xmlns=\"http://earth.google.com/kml/2.1\">\n");
    fprintf(out,"<Document><name>%s_%s</name><open>1</open>\n",Outfiletitle, type);
    fprintf(out,"<Style id=\"basePolyStyle\">\n <PolyStyle>\n  <color>bf99ccff</color>\n");
    fprintf(out,"  <colorMode>normal></colorMode>\n </PolyStyle>\n</Style>");
}
```

TextBox 28 The Build_Kml_Header routine

The attributes for each county's TIGER file are placed in a folder. This is accomplished by following the call to Build_Kml_Header with a simple file print statement (TextBox 29).

```
if (oldposition == 0)
        Build_Kml_Header(out, type);
    //write the county fips to make management in Google easier (eg, turn off all tracts
    // a county)
fprintf(out,"<Folder><name>%s</name><open>0</open>", Ptrdlg->m_strFips);
```

TextBox 29 Setting up the folder for each county's features

The next section of the code is where the features are constructed and written to the KML output file. This consists of five steps which are repeated for each area feature to be created.

1. Gather all the rings for a given area feature in the Polyatt data structure.
2. Determine if any rings are inner boundaries, i.e., that define holes.

3. Determine the centroid of each polygon.
4. Determine which outer boundary contains each hole.
5. Write the area feature in KML format, placing a pushpin at the centroid of the largest polygon in each feature.

TextBox 30 contains code for completing Step 1, i.e., reading the lines that make up a ring into the Polyatt data structure. The while loop at the top ensures that all area features in the partstemp file will be processed. You may recall from the Match_Poly_Lines_*polytype* discussion that there is one value of nparts for each feature. The read statement finds the number of parts (closed loops) in the current area feature. Inside the while loop, the memory for storing the number of points associated with each ring (partnpts) is allocated, as is the size of the Polyatt structure array needed to store all the rings. The for loop initializes the Area and the centroid X and Y values for the area feature. The last few lines in TextBox 30 simply allocate space for storing the points in the ith ring. Those values are then read in to that memory.

```
while (fread(&nparts,sizeof(int),1,partstemp) == 1)
{
    partnpts = (int *)calloc(nparts,sizeof(int));
    fread(partnpts,sizeof(int),nparts,headtemp);
    thepoly = (Polyatt *)calloc(sizeof(Polyatt),nparts);
    thecentroid.Area = 0.0;
    nouterpolys = 0;
    for (i = 0; i < nparts; i++)
    {
        thepoly[i].Area = 0.0;
        thepoly[i].X = 0.0;
        thepoly[i].Y = 0.0;
        thepoly[i].Ishole = false;
        thepoly[i].Containedin = -1;
        thepoly[i].Ptcount = partnpts[i];
        curpoints = (point_t *)calloc(partnpts[i],sizeof(point_t));
        fread(curpoints,sizeof(point_t),partnpts[i],ptstemp);
        thepoly[i].Curpoints = (point_t*)calloc(partnpts[i],sizeof(point_t));
```

TextBox 30 Reading all the rings for the current area feature

Steps 2 and 3. The Build_Poly_Kml routine uses algorithms found at the excellent website of Paul Brouke. The URL for this <u>highly recommended</u> site is http://local.wasp.uwa.edu.au/~pbourke/geometry/. Two algorithms found on that site are used here: “Polygon area and centroid calucation” and “Inside/outside polygon test”. Determining which rings are inner boundaries (i.e., they define holes) and finding the centroid of the largest outer boundary ring is accomplished by completing an area calculation. The area of a polygon is calculated by the formula

$$Area = \left| 0.5 \sum_{i=0}^{n-1} x_i y_{i+1} - x_{i+1} y_i \right|$$

When the absolute value of the area calculation is ignored, the sign of the above calculation also indicates if the ring is digitized in a clockwise (area is negative) or counterclockwise (area is positive) direction. Recall that the Match_Poly_Lines_*polytype* routines orient the lines in a ring so the current area feature is on the right of the line (Figure 12). This means that a ring with a positive area is an inner boundary. Put another way, it defines a hole.

Something else to note is that the term “area” is being used very loosely. Since the coordinates in TIGER are latitudes and longitude (spherical coordinates), the calculation is not the true area. However, all that is needed here is the relative sizes of the parts of a multipart area feature, so this usage of the term “area” is not too egregious.

A second calculation is also performed along with the area. That is determining the centroid of the polygon that has the largest area. The X and Y values of the centroid of a closed ring are calculated using the following formulae.

$$X = \left| \frac{1}{6A} \sum_{i=0}^{n-1} (x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i \right|$$

and

$$Y = \left| \frac{1}{6A} \sum_{i=0}^{n-1} (y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i \right|$$

where A is the area of the polygon (in absolute value).

Consider a polygon composed of several rings, such as in Figure 13, Case B. Two issues are worth noting. It is probably best to put the polygons that make up an area feature in a folder, much like the folder used to store the parts of Loudon County, TN (FIPSCODE 47105) in Figure 14. A second issue is where to put a pushpin for a multipart area feature. In TGR2KML, the pushpin, if the user chooses to create one, is put at the centroid of the polygon with the largest area (ignoring holes).

TextBox 31 contains the code for carrying out steps 2 and 3.

```
for ( int j = 0; j < partnpts[i]; j++)
    {//assign points to the simple polygon and calculate area and centroid
        if (j < partnpts[i] - 1)
        {
            tempval = (curpoints[j].x * curpoints[j+1].y) - (curpoints[j+1].x * curpoints[j].y);
            thepoly[i].Area += tempval;
            thepoly[i].X += (curpoints[j].x + curpoints[j+1].x) * (tempval);
            thepoly[i].Y += (curpoints[j].y + curpoints[j+1].y) * (tempval);
        }
        thepoly[i].Curpoints[j].x = curpoints[j].x;
        thepoly[i].Curpoints[j].y = curpoints[j].y;
    }
    free(curpoints);
    thepoly[i].Area *= 0.5;
    thepoly[i].X = thepoly[i].X/(6 * thepoly[i].Area);
    thepoly[i].Y = thepoly[i].Y/(6 * thepoly[i].Area);
    if (thepoly[i].Area > 0.0) //counterclockwise
        thepoly[i].Ishole = true;
    else
        nouterpolys++;
    //if area of current part is larger (in abs) than stored area and is not a hole (area < 0)
    if (thecentroid.Area > thepoly[i].Area)
    {
        thecentroid.Area = thepoly[i].Area;
        thecentroid.X = thepoly[i].X;
        thecentroid.Y = thepoly[i].Y;
    }
}//this closes the for loop on i at the top of TextBox 29
//when we get here, all the parts for the current census poly have been read in and we know
//which are holes
```

TextBox 31 Determining the areas and centroids of rings. A positive area indicates a hole.

In order to determine which hole belongs to which outer boundary, we use a point in polygon algorithm. For this algorithm we take a point on the boundary of a hole (a polygon with a positive area calculation) and find the outer polygon (polygon with a negative area calculation) that contains it. TGR2KML uses the first point in polygon algorithm on Paul Bourke’s web site. (His site has several versions.)

TextBox 32 contains the section of Build_Poly_Kml that assigns holes to their containing polygons. It makes a call to a routine named InsidePolygon. The code for that function is in TextBox 33.

```
//we now have to assign the holes to their containing polygons
int result;
for (i = 0; i < nparts; i++)
    {
    //find inside -- for each hole
    if (thepoly[i].Ishole)
        {
        for (int j = 0; j < nparts; j++)
        {
            if (i != j)
            {
            result = InsidePolygon(thepoly[j].Curpoints, thepoly[j].Ptcount, thepoly[i].Curpoints[0]);
            if (result == 0)
            {
                thepoly[i].Containedin = j;
                break;
            }
        }
    }
}
```

TextBox 32. Assigning holes to the polygons that contain them

The remainder of Build_Poly_Kml consists of writing each feature in the proper format. The name field is read from the current record that was pulled from the dbf.tmp file, and the attributes for each area feature are written in an html table. Pushpins, assuming the user wants them, are added to the KML file. Once the last feature is processed, the closing lines of the KML file are written and the file is closed. The code for accomplishing these tasks is straightforward and not repeated here.

Concluding Remarks

I have attempted to give the interested reader some guidance on how the information in TIGER is processed by TGR2KML. My hope is that students and others interested in pursuing GIS data programming can benefit from this effort. Users should be able to add line or point features, as I have left all the code for reading and preparing those features in the source files. One would simply need to change the output format from shape to KML.