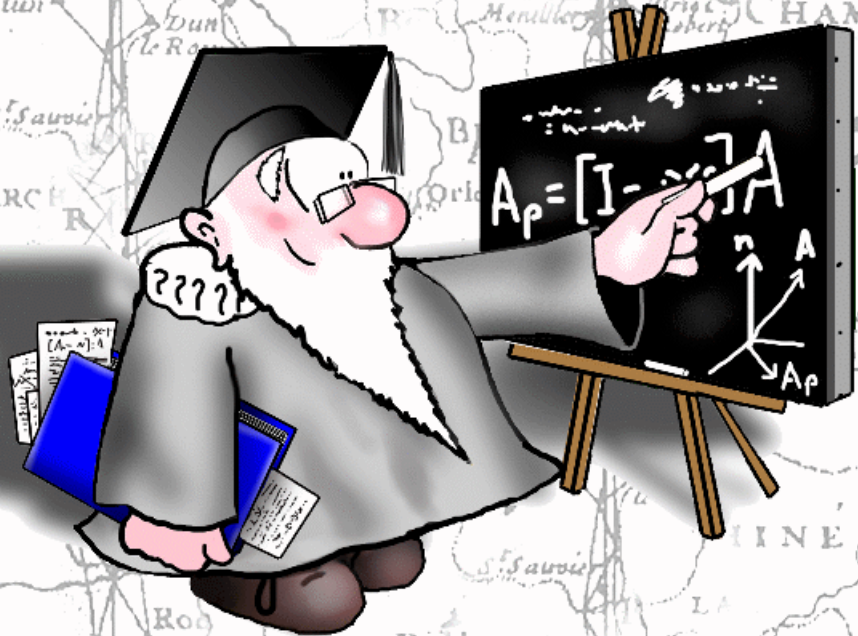


Getting Started



Writing Scripts with SML



in

TNTmips[®]

TNTedit[™]

TNTview[®]

Before Getting Started

This booklet introduces techniques for creating scripts in the Spatial Manipulation Language (SML) in the TNT products. The exercises in this booklet introduce you to concepts and techniques for writing powerful scripts for custom manipulations of the spatial data objects in your TNT Project Files.

Prerequisite Skills This booklet assumes that you have completed the exercises in *Getting Started: Displaying Geospatial Data* and *Getting Started: Navigating*. Please consult those booklets and the TNTmips Reference Manual for any review of essential skills and basic techniques you need. This booklet also assumes that you have at least a fundamental knowledge of one or more programming languages such as C, BASIC, or Pascal. You can begin to use SML even if you have no programming background, but SML is a powerful language and yields the most benefit in the hands of a good programmer.

Sample Data The exercises in this booklet use sample data that is distributed with the TNT products. If you do not have access to a TNT products CD, you can download the data from the MicroImages web site. In particular, this booklet uses scripts in the LITEDATA / SML data collection, the CUSTOM, MACRSCR, and TOOLSCR subdirectories under your primary TNT directory, and objects in the CB_DATA, SF_DATA, SURFMODL, and EDITRAST data collections. ***Install the sample files to your hard drive; you may encounter problems if you work directly with the sample data on the CD-ROM.***

More Documentation This booklet is intended only as an introduction to the Spatial Manipulation Language. Consult the TNT reference manual, and especially the online SML Reference for more information.

TNTmips and TNTlite® TNTmips comes in two versions: the professional version and the free TNTlite version. This booklet refers to both versions as “TNTmips.” If you did not purchase the professional version (which requires a software license key), TNTmips operates in TNTlite mode, which limits object size, and enables data sharing only with other copies of TNTlite. SML is not available in TNTatlas. All the exercises can be completed in TNTlite using the sample geodata provided.

Keith Ghormley and Randall B. Smith, Ph.D., 31 October 2001

©MicroImages, Inc., 1997

It may be difficult to identify the important points in some illustrations without a color copy of this booklet. You can print or read this booklet in color from MicroImages' web site. The web site is also your source for the newest Getting Started booklets on other topics. You can download an installation guide, sample data, and the latest version of TNTlite:

<http://www.microimages.com>

SML in the TNT Products

The Spatial Manipulation Language (SML) is a programming language that lets you write scripts that operate on the geospatial data objects in TNT Project Files. SML scripts can be executed from custom menus and icon bars, from an icon on your computer's desktop, or even from the operating system command line.

SML is a customization and design tool. With SML, you can use TNT products for tasks beyond the pre-defined processes found in the standard TNT menus. You can create simple processing scripts, or complete special-purpose products for a targeted private market. You can even bundle your scripts together with selected geodata objects in a Project File and distribute the whole package as a turn-key APPLIDAT (APPLIcation plus DATA).

MicroImages provides scores of sample SML scripts to give you models to work from. You can examine and adapt scripts ranging from simple processing routines to complex APPLIDATs that contain hundreds of lines of code. Your SML scripts will be easy to modify and enhance over the life of your project. You can quickly prototype and test program features.

STEPS

- select Install Sample SML Scripts from the TNT products CD installation menu

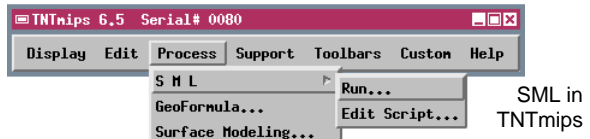
In later exercises, you will access them in the /CUSTOM subdirectories under your TNT products directory.

The exercises on pp. 4-18 introduce basic SML concepts and scripting conventions. Pages 19-27 illustrate specific program techniques for different types of geodata objects. The remainder of the book introduces advanced SML development techniques: building dialog windows; movie scripts; APPLIDATs; and Tool and Macro Scripts.

SML in an APPLIDAT



SML scripts are completely platform independent; they run without modification on any computer that runs the TNT products.



SML is available in TNTmips, TNTedit, and TNTview. In addition, all forms of scripts in the TNT products use constructs drawn from SML (with minor variations):

- queries and style scripts
- GeoFormulas
- CartoScripts
- APPLIDATs
- Tool Scripts
- Macro Scripts

Refer to the Getting Started booklets *Building and Using Queries*, *Using CartoScripts*, and *Using Geospatial Formulas* for information about particular script types.

Run VIEWSHED.SML

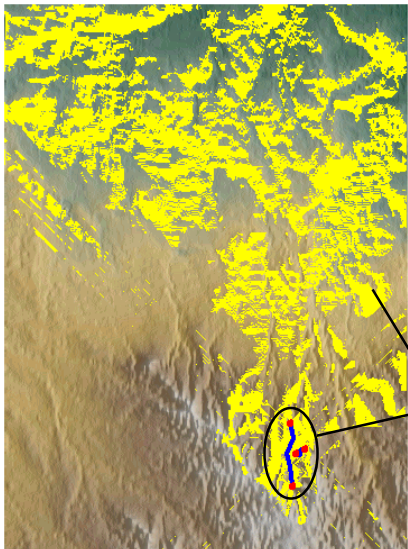
STEPS

- ☑ select Process / SML / Edit Script
- ☑ choose File / Open / *.SML File and select VIEWSHED.SML from the LITEDATA / SML folder
- ☑ scroll through the script for a first look at SML
- ☑ click [Run...] at the bottom of the window
- ☑ when prompted for the input raster "RIN", select ELEVATION from the CB_TM Project File in LITEDATA / CB_DATA
- ☑ for the input vector "V", select VROAD from the VIEWSHED Project File in LITEDATA / SML
- ☑ select a new Project File and object for the output raster "ROUT"
- ☑ use the display process to view three layers: CB_TM / ELEVATION, your new output raster, and VIEWSHED / VROADS

The VIEWSHED.SML script along with its sample data in VIEWSHED.RVC is contained on the TNT products CD-ROM, and is also available on the MicroImages web site. The script creates an output binary raster object that shows which parts of its input elevation surface are visible from the stream of points along the input line elements. Many applications that deal with line-of-sight surface characteristics can use the techniques illustrated in this script.

Start SML and load the VIEWSHED.SML script by following the steps listed. Before you run the script, scroll through it and survey its contents. Unless you are unfamiliar with a programming language such as C or BASIC, you should recognize statement forms and programming structures.

Note that the hardest work of the script is done with calls to various SML functions, such as `RasterToBinaryViewshed()`. MicroImages is constantly adding new functions to SML. Being aware of what functions are available and understanding what they do is essential to making the most of SML. In addition to using the built-in SML functions, you can write your own SML extensions in C with TNTsdk, and invoke external programs from within SML scripts (see page 28).



VIEWSHED.SML produces a binary raster (1's shown here in yellow) that indicates the areas visible on an elevation surface (shown here in pseudo-color) from a stream of points along input vector line elements (shown here in blue). Thus if the line elements represent roads, then the yellow areas define the vistas available to travelers on that road.

Fundamentals of Scripting

An SML script can be anything from a single statement to a long structured program with nested logical branching constructs and external program calls. There are few formal structural constraints.

You can use white space in almost any way. SML does not care if you use tabs or spaces to indent lines, or if you leave blank lines, or even if you break a statement in the middle and continue it on the next line. Thus VIEWSHED.SML has a function call that is broken in the middle of the argument list and continued on the next line:

```
MapToObject( georefR, xVector, yVector,
             Rin, rCol, rLine )
```

In the same way, statements can be continued across several lines. Select New from the File menu to clear the SML window. Then type in the short script illustrated here. The initial `clear()` statement erases the contents of the Console Window. The `var1 = 3` statement assigns the value 3 to the variable `var1`. The first `print()` statement outputs the value of `var1` to the Console Window.

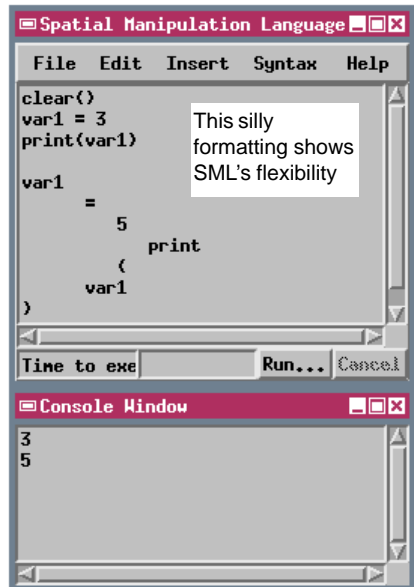
Note that the next three lines contain a single assignment statement: `var1 = 5`. The final `print()` statement is likewise distributed across four lines. Click [Run...] to execute the script.

Of course the silly formatting in this example is provided only to illustrate the flexible formatting SML supports. In your own scripts, use indents, spaces, and blank lines to enhance readability and to reflect the logical structure of the program. If you happen to use illegal syntax or formatting, SML does not execute the script when you click [Run], but posts an error message and puts the cursor in the script at the point of the error.

STEPS

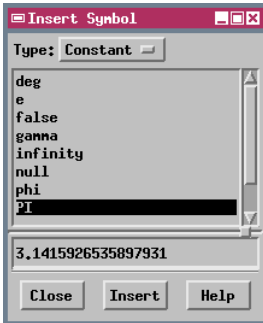
- clear the SML window by selecting New from the File menu
- type in the script illustrated below, using tabs or the space bar to indent the text
- select Syntax / Check to check the syntax
- click [Run] to execute the script

The **SML Window** is a simple text editor that provides access to function lists and syntax checking.



The **Console Window** shows the results of `print()` and other text input / output operations.

Variables and Constants

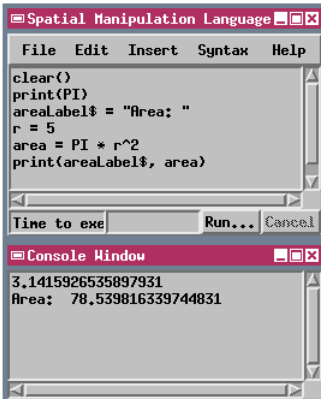


STEPS

- select File / New to clear the SML window
- type in the first two lines of the script in the form


```
clear()
print()
```

 and leave your cursor between the parentheses of the print() statement
- select Insert / Symbol, choose pi from the list, and click [Insert], [Close]
- type in the rest of the script shown below and click [Run]



Variables can be used for string, numeric, logical, array, class, and object (CAD, raster, vector, raster, region, and TIN) entities. Variables are created when the script first mentions them. With the exception of arrays and classes, they do not have to be declared ahead of time. Names can be up to 100 characters long and follow these conventions:

String: initial character is lowercase; must end in '\$' character. In the sample script on this page, `areaLabel$` is a string variable.

Numeric: initial character is lowercase; cannot end in '\$'. In the sample script on this page, `r` and `area` are numeric variables.

Object: initial character is uppercase
example: `GetInputRaster(R)`

Logical: implemented as numerics where 0 = false, and all non-zero values = true. Thus

```
done = 0;
if (condition) done = 1;
if (done) <statement>;
```

Array and Class names follow string and numeric conventions. You must declare an array or a class before using it. Enclose an array index in square brackets:

```
array numlist[10];
class COLOR red;
numlist[1] = 256;
```

You can use the Insert Symbol window (Insert / Symbol) to insert variables used previously in the script, or to insert predefined *constants* (whose values cannot be changed). Use the Type option menu to choose a variable type (or constant) and view the associated list. Inserting variable names rather than typing them can cut down on typing errors. Your variable names do not appear in these lists until you use the Check Syntax operation (see page 16) or run the script.

Expressions and Statements

Expressions are constructs that reduce to some value. Thus π^2 , 5.10, and $R[i,j]/100$ are all expressions. Expressions can be used on the right side of assignment statements and as arguments in function calls.

Statements can be simple or complex. A simple statement can consist of an assignment, such as

```
area = pi * r^2;
```

A *complex statement* is bracketed by the keywords “begin” and “end” in the form

```
if (condition) begin
    function(r);
    area = pi * r^2;
end
```

SML also lets you use braces (“curly brackets”) instead of spelling out “begin” and “end”:

```
if (condition) {
    function(r);
    area = pi * r^2;
}
```

Conditional statements have the form

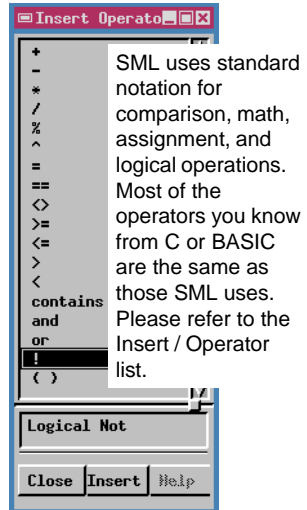
```
if (<condition>) then <statement>
else <statement>;
```

The *else* clause is optional, as is the “then”:

```
if (<condition>) <statement>;
```

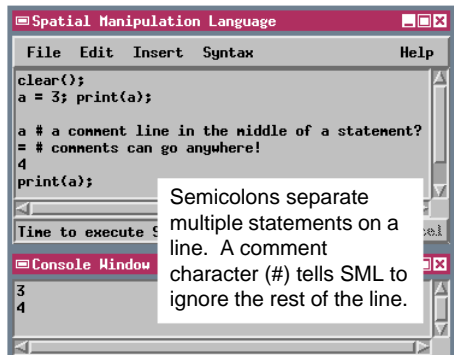
It is good practice (though optional) to use the terminator character (the semicolon, “;”) to mark the end of a statement. Using a terminator also lets you put multiple statements on a line, separating them with semicolons.

The comment character (“#”) tells SML to ignore the rest of the line. If a comment character is the first character on a line, SML ignores the whole line. You can also put a comment on the same line with other SML tokens, as long as the tokens come before the comment.



STEPS

- select File / Open / *.SML File and select LITEDATA / SML / EXPRESS.SML
- run the script
- change the area threshold for the if condition and run the script again



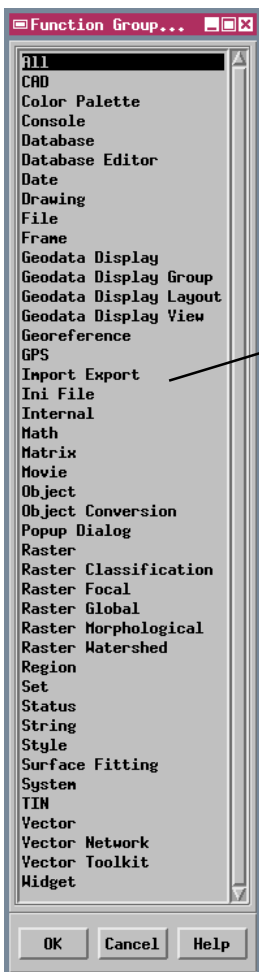
Built-in Functions

STEPS

- clear the SML window with File / New
- select Insert / Function
- click the Function Group button and browse the function library for each category

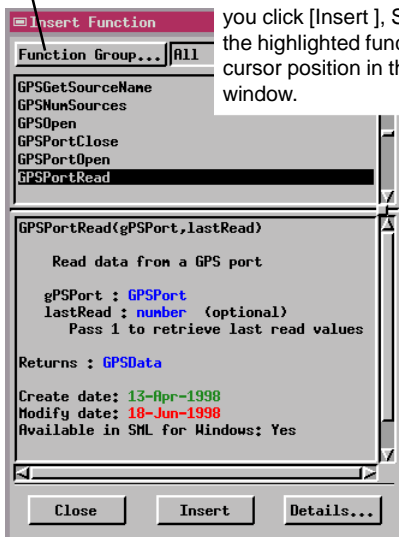
The real power of SML lies in its rich function library that lets you create, read, and write geospatial objects and subobjects in your TNT Project Files. Standard math functions are included along with specialized functions for display, interface, and data manipulation. MicroImages is constantly enhancing and expanding the SML functions to give you more ways to work with your geospatial data.

Select Insert / Function to open the Insert Function window, which lets you select functions and see their usage format specifications. Click the Function Group button to examine the available functions for each category. As you scroll through the list of functions, the definition in the lower pane changes to show the usage of the current function. Click the Insert button to copy the function into the SML script window.



The Function Group button opens a scrolling list of function categories.

The Insert Function window offers a scrolling list of functions in the top pane, and a function definition in the bottom pane. If you click [Insert], SML inserts the highlighted function at the cursor position in the SML window.



Online Function Help

The supporting documentation for SML functions is incorporated into the process. First, the bottom pane of the Insert Function window gives a simple definition, showing each argument and its data type. You can click the Insert button to copy a complete instance of the function into the SML window.

For more information, click the Details button in the Insert Function window. SML opens the Details On:

window that gives complete details, plus a working section of code that shows how the function works in a sequence of statements. You can click the Insert Sample button to copy the entire example into the SML window.

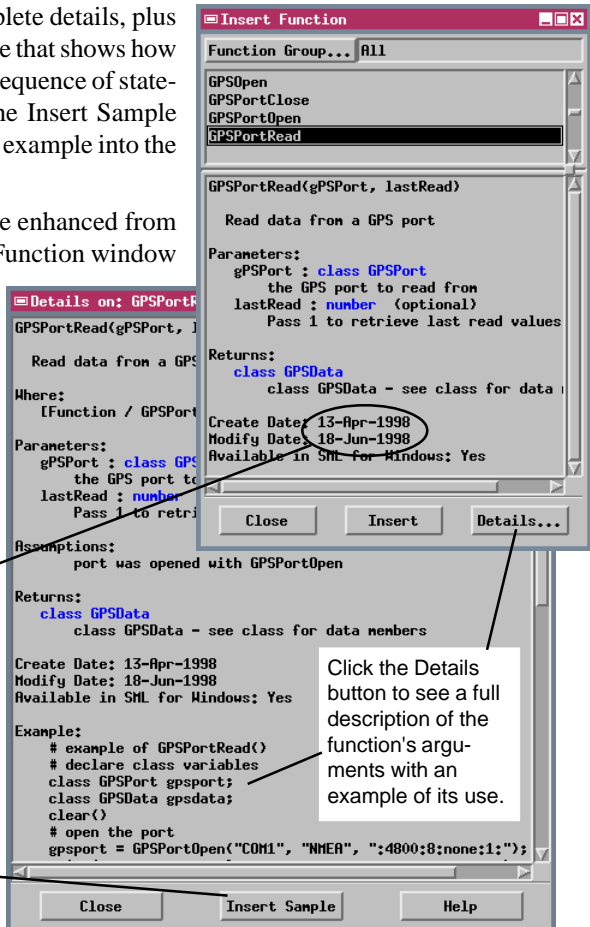
Since SML functions are enhanced from time to time, the Insert Function window shows when the current function was most recently changed. Watch for modifications that provide optional new capabilities to functions you have used.

The Create date tells when the function was introduced to SML. The Modify date tells when the function was last updated. Sometimes optional arguments are added to a function to expand its capabilities.

Click Insert Sample to copy the entire section of sample code into the SML window

STEPS

- select All in the Function Group text box
- scroll to the GPSPortRead() function
- click the Details button
- click [Insert Sample]
- examine the newly-inserted script lines in the SML script window

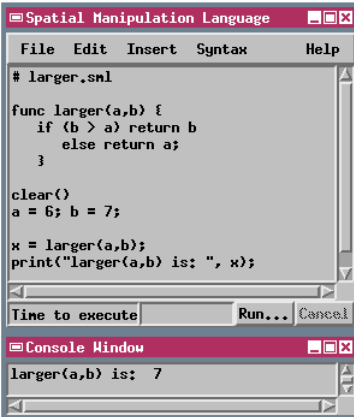


- close the Details and Insert Function windows when you have completed this exercise

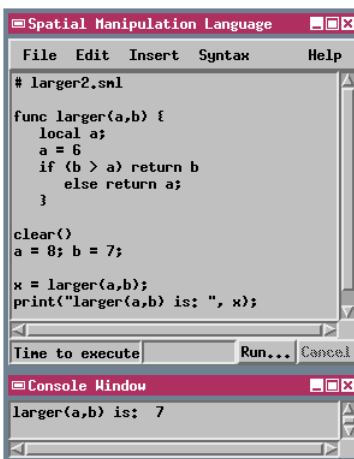
User-Defined Functions and Procedures

STEPS

- select File / Open / *.SML File and open LARGER.SML from the LITADATA / SML folder
- run the script



- select File / Open / *.SML File and open LARGER2.SML from the LITADATA / SML folder
- run the script



SML allows you to define your own functions and procedures that you can use to encapsulate sequences of program steps that must be repeated in several places in the script. User-defined functions must return a value, whereas procedures do not. Of course you must declare a function or a procedure before you invoke it, using the form:

```
func funcname ([parmlist])
  { statement; statement; ...
    return expr }
proc procname ([parmlist])
  { statement; statement; ... }
```

The simple example used in this exercise finds the larger of two values.

Unless declared otherwise, all script variables are global. This means that your functions and procedures can use and modify variables defined elsewhere in the script. Any global

variables and classes used in functions and procedures must be declared before the function definitions. In a large or complex script, this global scope of variables may cause unanticipated consequences. To limit the scope of a variable to a particular function or procedure, you must declare the variable as local variable within the function definition:

```
local x;
```

where x is a variable name. Local variables can have the same names as global variables elsewhere in the script, though this is not recommended practice. The script LARGER2 declares a local variable "a" within the definition of the function larger(a,b) and assigns it a value of 6. In the main part of the script, a global variable with the same name is declared and assigned a value of 8. As the result of the script illustrates, the global variable is ignored and the local value is used instead by the function.

Using Classes

A Class is a complex variable that consists of multiple members in the same way that a database record consists of multiple fields. A class variable may have any number of members and the members may be of any data types, including other classes.

Class variables are designed for passing information to and from complex functions. In many cases, the members of a class variable are set only by a function call, and so are read-only from the script's point of view; they cannot be given new values by assignment statements.

A class must be declared with the class keyword, in the form:

```
class COLOR background
```

which declares background to be a class variable of the Color type. Members of a class are specified in the form name.member (just as database values are specified in the form table.field). For example, the class Color has five members that can be assigned values with statements in the form:

```
background.red=50
background.green=75
background.blue=20
background.transp=0
```

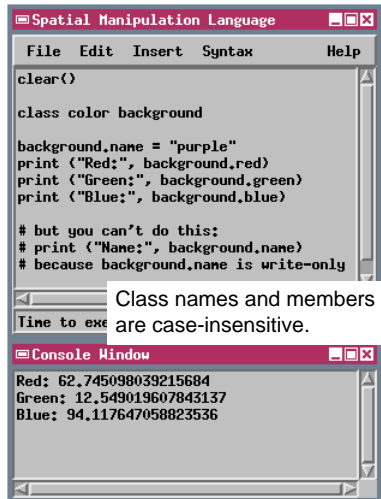
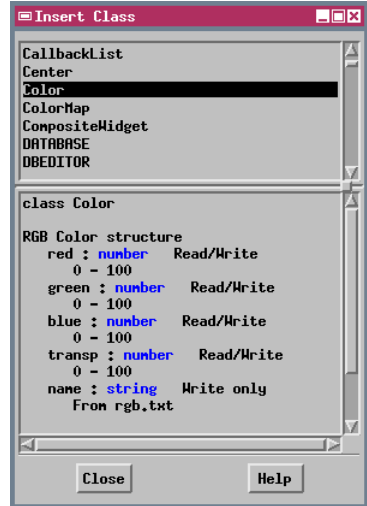
The name member of the Color class is used only to pass red, green, and blue values to the class variable from the standard reference file RGB.TXT. Thus

```
background.name = "purple"
```

sets the RGB components of the class variable background according to the definition of "purple" in RGB.TXT. The name member is write-only and cannot be read in other parts of the script.

STEPS

- clear the SML window with File / New
- select Insert / Class
- scroll through the list in the top pane of the Insert Class window and select class Color



Member Inheritance and Type Checking

STEPS

- ☑ select Insert / Class
- ☑ select POINT2D in the top panel of the Insert Class window, and examine its members
- ☑ select POINT3D in the top panel of the Insert Class window, and examine its members
- ☑ select XmPushButton in the top panel of the Insert Class window
- ☑ trace the line of class and member derivation shown in the bottom panel

Series of derived classes like those shown below are used in SML to represent the X Window / Motif structures used to create interface windows. All interface components are categories of a basic component called a *widget*.

An important concept with classes is *inheritance*. Class POINT2D represents the location of a 2-dimensional point; its members are the x and y coordinates of the point. Class POINT3D is said to be *derived* from class POINT2D. This means that a class variable you declare as POINT3D not only has its own member z, but also *inherits* members x and y from class POINT2D. You can use inherited members of a class in the same way you would its native members.

The use of classes allows *strong type checking*. Thus, when you invoke a function that wants a POINT2D for a parameter, you can pass it any POINT2D (or derivative class). But the function will refuse any variable that is not a POINT2D. For example, you could not pass such a function a Color class, because Color is not a POINT2D. By contrast, since POINT3D is derived from POINT2D, you could pass a POINT3D or anything else derived from POINT2D to a function that requires a POINT2D.

```
class XmPushButton
Simple push button widget
  ActivateCallback : class XmCallbackList  Read only
  ArmCallback : class XmCallbackList  Read only
  DefaultButtonShadowThickness : number  Read/Write
  FillOnArm : number  Read/Write
  ShowAsDefault : number  Read/Write
  MultiClick : string  Read/Write
  Possible values:
    "MULTICLICK_DISCARD"
    "MULTICLICK_KEEP"
```

The XmLabel and XmPushButton definitions show shared inheritance.

class XmPushButton is derived from class XmLabel and inherits the following members from it

```
Alignment : string  Read/Write
Possible values:
  "ALIGNMENT_BEGINNING"
  "ALIGNMENT_CENTER"
  "ALIGNMENT_END"
```

```
class XmLabel
A static text label; also base class of buttons
Alignment : string  Read/Write
Possible values:
  "ALIGNMENT_BEGINNING"
  "ALIGNMENT_CENTER"
  "ALIGNMENT_END"
```

```
LabelString : string  Read/Write
MarginBottom : number  Read/Write
MarginHeight : number  Read/Write
MarginLeft : number  Read/Write
MarginRight : number  Read/Write
MarginTop : number  Read/Write
MarginWidth : number  Read/Write
RecomputeSize : number  Read/Write
```

```
LabelString : string  Read/Write
MarginBottom : number  Read/Write
MarginHeight : number  Read/Write
MarginLeft : number  Read/Write
MarginRight : number  Read/Write
MarginTop : number  Read/Write
MarginWidth : number  Read/Write
RecomputeSize : number  Read/Write
```

class XmLabel is derived from class XmPrimitive and inherits the following members from it

```
ButtonShadowPixel : number  Read/Write
ButtonShadowColor : class COLOR  Write only
ForegroundPixel : number  Read/Write
ForegroundColor : class COLOR  Write only
HighlightPixel : number  Read/Write
HighlightColor : class COLOR  Write only
HighlightOnEnter : number  Read/Write
HighlightThickness : number  Read/Write
```

Follow the inheritance from Widget to XmPrimitive to XmLabel to XmPushButton.

Class Methods

Some classes include their own functions and procedures, which are collectively called class methods. Class methods may be used to pass values into a class or to perform some other operation related to the class. Class methods are invoked using the form `name.method()`, where `name` is the name of the class variable.

Class `VIEWPOINT3D` represents the settings for 3D rendering in a 3D View window. It includes member `ViewPos`, a `POINT3D` class variable that holds the `x`, `y`, and `z` coordinates of the viewer. A class method is used to pass the required values into the class:

```
Class VIEWPOINT3D vp;
Class POINT3D vpos;
vpos.x = 523487;
vpos.y = 1473245;
vpos.z = 2000;
vp.SetViewerPosition(vpos);
```

This class method is a procedure, and so does not return a value.

The methods in the `STRING` class are all functions that return either a string or a numerical value. Try typing in and running the following example:

```
clear();
Class STRING txt$;
txt$ = "watershed";

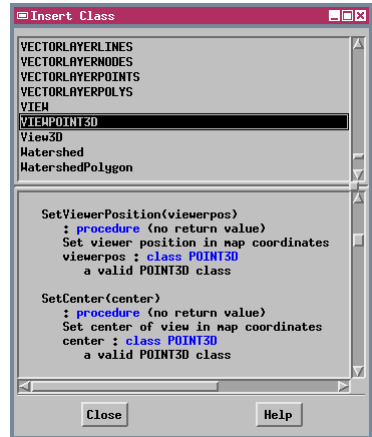
char1$ = txt$.charAt(1);
print(char1$);

uc$ = txt$.toUpperCase();
print(uc$);
```

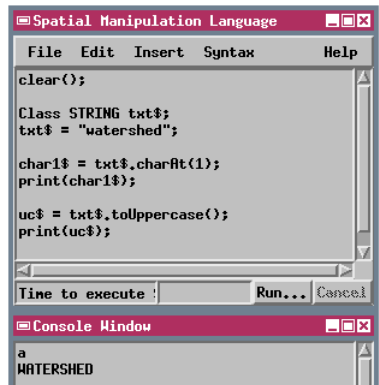
The `charAt(n)` method returns the `n`'th character in the string (indexed with the leftmost character at 0). The `toUpperCase()` method returns a copy of the string in all uppercase characters.

STEPS

- select Insert / Class
- select `VIEWPOINT3D` in the top panel of the Insert Class window
- scroll the bottom panel and examine the class methods



- select `STRING` in the top panel of the Insert Class window
- scroll the bottom panel and examine the class methods



User Input

STEPS

- ☑ clear the SML window with File / New
- ☑ type in the console window prompt and input statements illustrated and [Run] the script
- ☑ choose Insert / Function
- ☑ click the Function Group button, select Popup Dialog from the Function Group window and click [OK]
- ☑ choose File / Open and select LITEDATA / SML / POPUP.SML and [Run] the script

The simplest type of user input and output uses the console window. You can print prompt strings and capture user responses using the `print()` and `input$()` functions. The console input code is simple for the author of the script, but console prompts may be missed by an inattentive user.

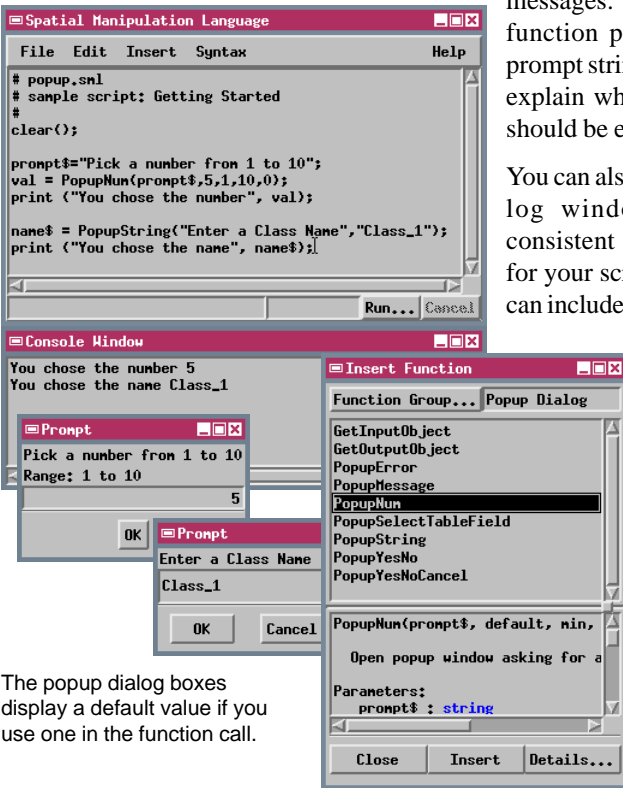
```
clear(); print("Enter your name:")
name$ = input$()
print("Your name is: ",name$)
```

Popup dialog windows offer more flexibility and at the same time are less likely to confuse the user. SML includes predefined functions in the Popup Dialog function group that open dialogs for input of numeric or string values, yes-no responses, and display error

messages. Where required, the function parameters include a prompt string that you can use to explain what value or response should be entered by the user.

You can also build your own dialog windows to provide a consistent interactive interface for your script. These windows can include push buttons, menus,

lists, and other components that you are familiar with in the TNTmips user interface. Samples showing how to created SML dialog windows are included later in this booklet.



The popup dialog boxes display a default value if you use one in the function call.

Loops and Branches

Implied Loops. When SML sees a raster object variable on the left side of an assignment statement, it executes an implied loop, evaluating the right side of the statement and assigning the result to each cell in the left-side raster object:

```
R = R * scale # multiplies each cell in R
```

For each statements for raster and vector objects have the forms:

```
for each Rastvar statement
for each Rastvar[lin,col] statement
for each Rastvar in Region statement
for each vector_element[n] in V statement
```

In the raster (R) notation, `lin` and `col` indicate the line number and column number of the "current position" in the raster for access within the processing loop. In the vector (V) notation, `vector_element` can be "point", "line", "poly", or "node". The `n` is optional and can be omitted. If given, the variable `n` is used as the loop counter.

For statements have two forms:

```
for var=expr to expr statement
for var=expr to expr step expr statement
```

Loops using "for" statements allow a script to operate on portions of a set of values (raster cells, array values, element numbers) specified by ranges, or to "step" through a set of values.

While. Be careful of "while" loops.

```
while (condition) statement
```

As long as the loop condition tests true, the loop continues. If the condition never becomes false, you get an infinite loop.

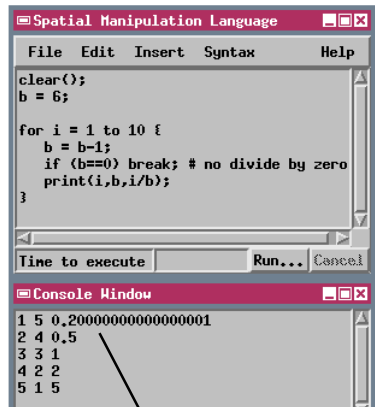
```
a = 0;
while (a <= 360) {
  print (a, sin(a/deg));
  a = a + 1;
}
```

STEPS

- select File / Open and select WHILEFOR.SML from the LITEDATA / SML folder
- run the script
- change the while condition and run the script again
- change the step value in the for loop and run the script again

NOTE: the "for each" keyword sequence also may be written as one word: "foreach". This version of SML does not support nested "for each" commands.

The **break** statement is used to exit a loop before the loop might otherwise terminate. It is often used in a conditional test inside the loop. The break statement in this example prevents divide by zero.



Notice that as with all computer systems, some operations yield very small errors in floating point values (1 / 5 yields 0.200000000000000001).

Script Development and Checking

STEPS (not for Macintosh)

- keep the script from the previous exercise open
- open another instance of the SML window with Process / SML / Edit Script
- move the new SML window so that it does not obscure the first one
- select the code for the "while" loop from the WHILEFOR script
- use the Copy and Paste selections on the Edit menus to copy the selected section to your new script
- remove the closing "]" character from your new script
- choose Syntax / Check for the new script
- choose File / Exit for the new script window and do not save changes

The easiest way to develop an SML script is to adapt the sample scripts distributed with the TNT products to your own needs. You can open two SML script editing windows side by side and use the copy and paste menu functions to copy sections of code from the MicroImages sample script into the script you are developing. If you are running under a Windows operating system, the SML cut and paste functions use the Windows clipboard, so you can also cut and paste text between the SML editor and an editor running under Windows.

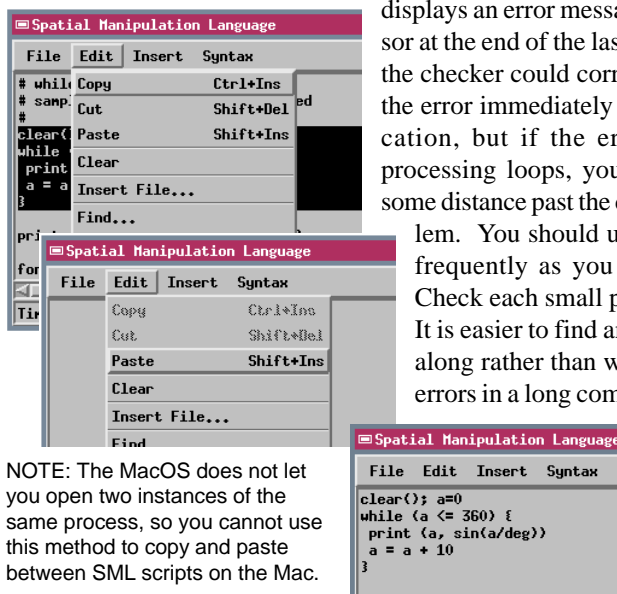
The Check option on the Syntax menu checks your script for syntax errors. The types of errors that can be found include missing function parameters, function and variable misspellings, and unclosed parentheses and loops. The syntax checker cannot detect logical errors such as infinite loops or incorrect input values.

If the syntax checker finds problems in your script, the message line at the bottom of the SML window

displays an error message and places the cursor at the end of the last part of the script that the checker could correctly interpret. Often the error immediately follows the cursor location, but if the error involves nested processing loops, you may need to search some distance past the cursor to find the problem.

You should use the syntax checker frequently as you develop your script. Check each small portion as you write it. It is easier to find and fix errors as you go along rather than waiting to fix all of the errors in a long complex script. Checking

syntax also updates the lists of variables available for insertion from the Insert / Symbol window.



NOTE: The MacOS does not let you open two instances of the same process, so you cannot use this method to copy and paste between SML scripts on the Mac.

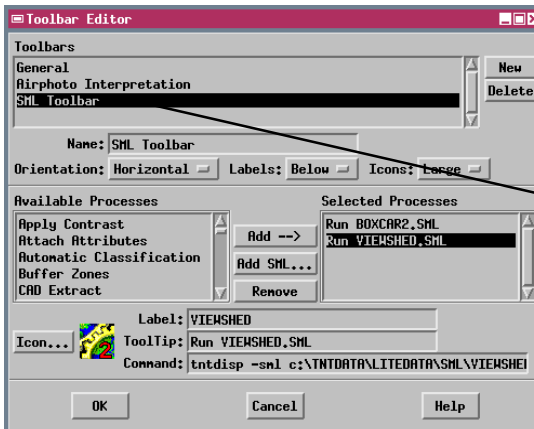
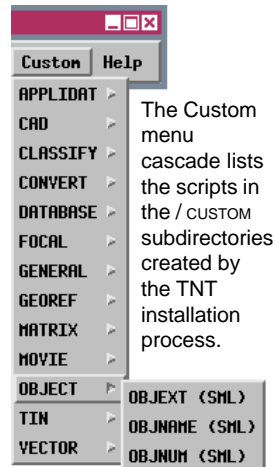
Toolbars and the SML Custom Menu

You can select and run any SML script without opening the SML editor window by selecting SML / Run from the Process menu. You can also assign SML scripts to icons on custom toolbars. Use the Toolbar Editor window to create or select a toolbar, set a horizontal or vertical orientation, and set up label positions. Then select one or more SML scripts and edit the Label and Tooltip text boxes as illustrated to establish the interface text for each. Press the Icon button to select an icon for each script. The steps in this exercise create a new SML toolbar with two script icon buttons.

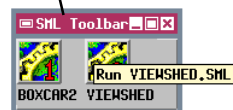
Sample SML scripts provided by MicroImages also can be run from the Custom menu cascade if you have installed them from the TNT products CD. (If your TNT menu bar does not have a Custom menu, run the TNT setup program from the CD and select **Install Sample SML Scripts**.) The setup program creates a /CUSTOM subdirectory and copies the sample *.SML scripts to it. Thereafter, any new scripts that you put into the /CUSTOM directory also become available on the Custom menu.

STEPS

- choose Toolbars / Edit in the TNTmips main menu
- press [New] in the Toolbar Editor window
- edit the Name field to read "SML Toolbar"
- select Horizontal from the Orientation menu
- click [Add SML...]
- select BOXCAR2.SML
- click [Icon...] and select an icon
- repeat the previous two steps for VIEWSHED.SML
- click [OK] to finish



Use the Toolbar Editor to add BOXCAR2 and VIEWSHED icons to a new SML toolbar.



Script Objects and Encryption

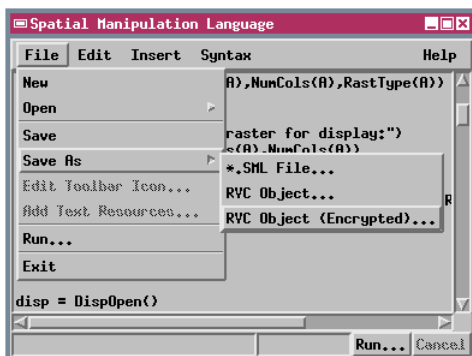
STEPS

- select File / Open / *.SML File and choose /LITEDATA / SML / BOXCAR2.SML
- select File / Save As / RVC Object (Encrypted)
- create a new Project File and SML object as prompted
- select an encryption password in the Encryption Options window
- if you are not using TNTlite, use File / Open to select your encrypted script (it shows only an encryption message)

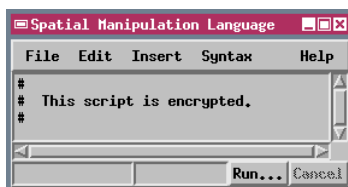
NOTE: A license key is required to run an encrypted SML script object. Thus encrypted scripts cannot be run in TNTlite.

So far you have worked with SML scripts that have been saved as independent text files with the SML file extension. These are 1-byte text files that can be opened with any text editor. If you do edit a script file with another editor, be sure to save it with the SML extension.

An SML script also can be saved as a script object in a Project File (use File / Save As / RVC Object). This allows you to put input, output, and script objects all in the same file if you find this more convenient. Another advantage to storing a script in a Project File is the ability to **encrypt** a script object. You may want to distribute your scripts to others but still protect your development efforts and proprietary algorithms. An encrypted script object can only be run by authorized TNTmips users and cannot be viewed or edited by anyone (including the creator; always keep an unencrypted copy of the script for reference or further development). You can allow an encrypted script to be run by any TNTmips user or limit its use to computers with a specific software license key number. You can also choose to require a password for running the script.



Use the Save As / Encrypted option to create an encrypted copy of the script in a Project File. If you open an encrypted script in the SML window, it shows only an encryption message. **IMPORTANT: Always keep an unencrypted copy for editing.**



Raster Objects

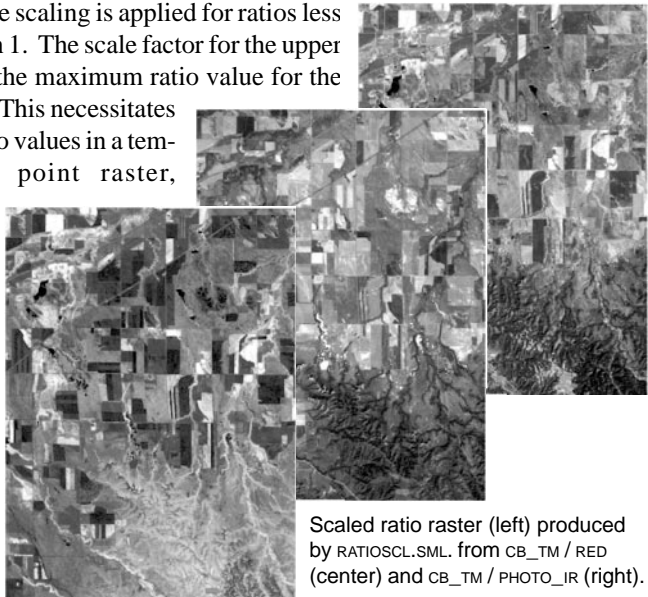
A full set of raster functions let your SML scripts read, create, and analyze raster objects. You can write mathematical expressions to compute values for a new raster object from one or more input rasters or use various higher-level SML functions to create new raster values.

Use the `GetOutputRaster()` and `CreateRaster()` functions to create new raster objects. When you create an output raster object, give some thought to your choice of the specifics of its data type: binary, integer, signed, unsigned, and floating point. For example, if your script's computations can create negative output cell values, be sure to specify a signed data type. Several functions provide access to raster subobjects.

The `RATIOSCL` sample script is designed to compute the ratio between two raster image bands (assumed to be 8-bit unsigned rasters) and rescale the result to the 8-bit unsigned data range for the output raster. The raw ratio values could range from $.004 (1 / 255)$ to 255, and separate scaling is applied for ratios less than or greater than 1. The scale factor for the upper range is based on the maximum ratio value for the entire image area. This necessitates storing the raw ratio values in a temporary floating point raster, computing the scale factor from the maximum ratio value, then computing the rescaled values and writing them to the final output raster.

STEPS

- select File / Open and select `RATIOSCL.SML` from the `LITEDATA / SML` folder
- study the script structure and statement syntax
- run the script
- when prompted for a raster for `N`, select `PHOTO_IR` from the `CB_TM` Project File in `LITEDATA / CB_DATA`
- select `RED` from the `CB_TM` Project File for input object `D`
- create a new raster object for `RS`
- for this exercise and those on the following pages, use the Display process to display the input object(s) and the new object created by the script



Scaled ratio raster (left) produced by `RATIOSCL.SML` from `CB_TM / RED` (center) and `CB_TM / PHOTO_IR` (right).

Vector Objects

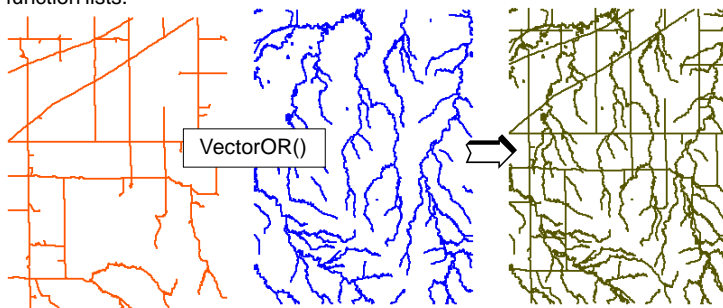
STEPS

- ☑ select File / Open / *.SML File and open the script VECTOCOMB.SML from LITEDATA / SML
- ☑ run the script using for input HYDROLOGY and ROADS from CB_DATA / CB_DLG

```

CloseVector
CreateTempVector
CreateVector
FindClosestLabel
FindClosestLine
FindClosestNode
FindClosestPoint
FindClosestPoly
GetInputVector
GetInputVectorList
GetOutputVector
GetVectorLinePointList
GetVectorNodeLineList
GetVectorPolyAdjacentPolyList
GetVectorPolyIslandList
GetVectorPolyLineList
NumVectorLabels
NumVectorLines
NumVectorNodes
NumVectorPoints
NumVectorPolys
OpenInputVectorList
OpenVector
VectMerge
VectorAND
VectorElementInRegion
VectorExists
VectorExtract
VectorOR
VectorReplace
VectorSubtract
VectorToolkitInit
VectorXOR
    
```

Vector functions are listed in the Vector (above), Vector Network, and Vector Toolkit function lists.



The short script shown above uses VectorOR() to combine two input vector objects into a single output vector object.

A growing list of functions support vector object creation, reading, writing, and manipulation. Look for vector function definitions in the Vector, Vector Network, and Vector Toolkit groups.

A simple script illustrates basic functions for input, output, and one of the vector combinations:

```

GetInputVector(Voperator);
GetInputVector(Vsource);
GetOutputVector(Vor);
Vor = VectorOR(Voperator, Vsource);
    
```

Vector extraction operations are supported by similar functions. For an example, refer to the sample script CUSTOM / VECTOR / VECINTR.SML.

SML also supports more complex interaction between vector objects and objects of other types. You have already seen VIEWSHED.SML (page 4). Another example is provided in CUSTOM / FOCAL / VECFOCAL.SML, which uses points in a vector object to select cells in a raster object and applies the FocalMean() function to each of those cells in turn. Open that script and observe how the vector coordinates ($x=V.point[i].Internal.x$) are translated into map coordinates using the georeference function ObjectToMap(V,x,y,georefV,xVector,yVector), and how MapToObject(georefR, xVector, yVector, R, rCol, rLine) finds the raster cell corresponding to the map coordinates.

Using the Vector Toolkit

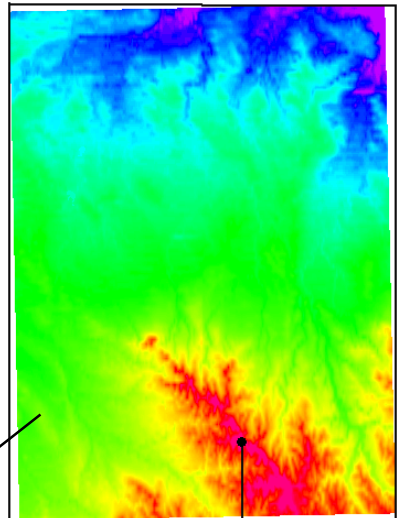
The functions in the Vector Toolkit function group enable a script to modify elements in an existing vector object or add new elements to an object. To modify an existing vector object, the script must first initialize the vector toolkit for use with that object:

```
GetInputVector(V);
VectorToolkitInit(V);
    [Editing operations with vector
     toolkit functions]
CloseVector(V);
```

When you will be adding elements to a new output vector object, toolkit initialization can be done when the object is created. The second argument to the `GetOutputVector()` function is an optional flag string that can be used to set the topology level and to initialize the vector toolkit. For example, setting this argument to "VectorToolkit,Polygonal" initializes the vector toolkit and establishes polygonal topology for the vector object.

The sample script `VTOOLKIT.SML` shows how some of the vector toolkit functions can be used to create elements in a new vector object. The script first opens an input raster and finds its geographic extents and the map position of the cell with the highest value. The script then creates a new vector object with implied georeference to the input raster object, adds a point element at the position of the maximum cell value, and draws a vector line outlining the raster extents. The location on this boundary line that is closest to the maximum cell point is then found, and a line is added connecting these two locations. The vector object is then validated (to check topology and compute standard attributes) and closed.

Raster `DEM16_BIT` and the vector object created from it by the sample script.



STEPS

- select File / Open / *.SML File and open the script `VTOOLKIT.SML` from `LITEDATA / SML`
- study the script structure and comments
- run the script using for input `DEM_16BIT` from the `CB_ELEV Project File` in `LITEDATA / CB_DATA`

```
ClosestPointOnLine
VectorAddLabel
VectorAddLine
VectorAddNode
VectorAddPoint
VectorAddTwoPointLine
VectorChangeLine
VectorChangePoint
VectorDeleteAngleLines
VectorDeleteLabel
VectorDeleteLabels
VectorDeleteLine
VectorDeleteLines
VectorDeleteNode
VectorDeleteNodes
VectorDeletePoint
VectorDeletePoints
VectorDeletePoly
VectorDeletePolys
VectorDeleteStdAttributes
VectorLineRayIntersection
VectorSetFlags
VectorSetZValue
VectorUpdateStdAttributes
VectorValidate
```

CAD and TIN Objects

STEPS

- ☑ select File / Open / *.SML File and open the script CAD.SML from LITEDATA / SML
- ☑ examine and then run the script using raster object HAYWARD from the HAYWDEM Project File in LITEDATA / SF_DATA
- ☑ open the script TIN.SML from LITEDATA / SML
- ☑ study and then run the script, using object ELEV_PTS from the SURFACE Project File in LITEDATA / SURFMDL for the input

```

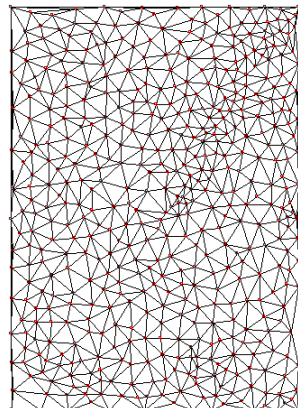
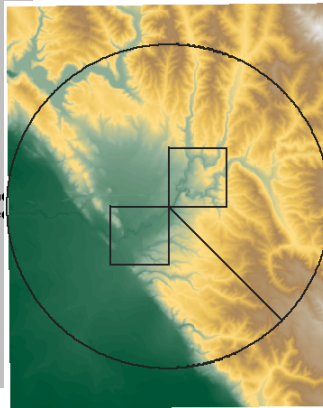
CADAttachDBRecord
CADCreateBlock
CADElementInRegion
CADElementType
CADGetElementList
CADInsertBlock
CADNumBlocks
CADNumElements
CADReadArc
CADReadArcChord
CADReadArcHedge
CADReadBox
CADReadCircle
CADReadEllipse
CADReadEllipticalArc
CADReadEllipticalArcChord
CADReadEllipticalArcHedge
CADReadLine
CADReadPoint
CADReadPoly
CADReadText
CADUnattachDBRecord
CADWriteArc
CADWriteArcChord
CADWriteArcHedge
CADWriteBox
CADWriteCircle
CADWriteEllipse
CADWriteEllipticalArc
CADWriteEllipticalArcChord
CADWriteEllipticalArcHedge
CADWriteLine
CADWritePoint
CADWritePoly
CADWriteText
CloseCAD
CreateCAD
GetInputCAD
GetOutputCAD
OpenCAD
    
```

A growing list of functions support CAD and TIN object creation, reading, writing, and manipulation. Sample script CAD.SML uses some of the numerous CAD functions. The script uses a raster object as input to define geographic extents and georeferencing and creates a new georeferenced CAD object to which several elements are added. A circle element is drawn centered at the geographic center of the raster, then a line element is drawn from the center to the circumference of the circle. Several box elements are then added around the center point.

Sample script TIN.SML illustrates some of the TIN functions. It uses the TINCreateFromNodes() function to make a new TIN object from arrays of node coordinates. The coordinate arrays are created in this case by reading the coordinates of points in a 3D vector object. The script also uses functions to read the number of TIN hulls, edges, and triangles.

```

CloseTIN
GetInputTIN
GetOutputTIN
TINAddNode
TINCreateFromNodes
TINDeleteEdgeAndMakeHole
TINDeleteNode
TINDeleteNodeAndMakeHole
TINDeleteTriangleAndMakeHole
TINDeleteTrianglesInPolygon
TINElementInRegion
TINGetConnectedEdgeList
TINGetConnectedNodeList
TINGetEdgeExtents
TINGetEdgeNodesAndTriangles
TINGetNodeExtents
TINGetNodeZValue
TINGetSurroundTriangleList
TINGetTriangleExtents
TINGetTriangleNodesAndEdges
TINGetTrianglesInPolygon
TINNumberEdges
TINNumberHulls
TINNumberNodes
TINNumberTriangles
TINSetNodeZValue
    
```



Region Objects

You can also create and use region objects in SML scripts. Region objects represent the outline of a region of interest in operations on other spatial objects. SML functions in the Region function group allow you to open and save region objects, check if particular map coordinates lie within the region, and perform region combination operations (AND, OR, Subtract, and XOR). Several functions in the Object Conversion group allow you to convert vector and binary raster objects into region objects.

SML provides a simple way to use a region object to restrict actions on a raster object. The simple construction

```
for each RastVar in RegionVar {
  [actions]
}
```

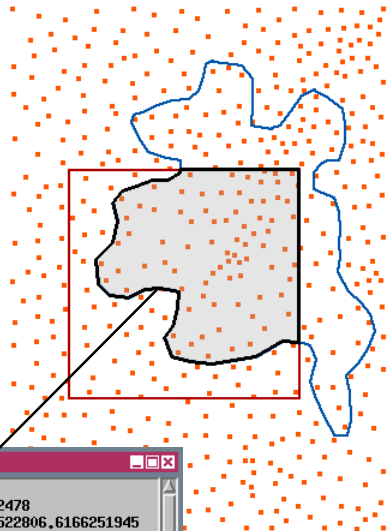
restricts the actions to raster cells that lie within the region boundaries. This construction provides a simpler alternative to using values in a binary mask raster to control the operations.

The sample script REGION.SML illustrates the use of some of the region functions. The script opens two region objects and uses the RegionAND() function to find the region that is their intersection. This new region is then used to find information about point elements in the corresponding area of an input 3D vector object. The script uses the PointInRegion() function in a "for each" loop to examine each point's coordinates and select only those points that lie within the region.

STEPS

- select File / Open / *.SML File and open the script REGION.SML from LITEDATA / SML
- study the script structure and comments
- run the script using for input the region objects POLYREGION and RECTANGLE from the REGION Project File in LITEDATA / SML and vector object ELEV_PTS from the SURFACE Project File in LITEDATA / SURFMODL.

```
ClearRegion
CopyRegion
CreateRegion
GetInputRegion
GetOutputRegion
OpenRegion
PointInRegion
RegionAND
RegionOR
RegionSubtract
RegionTrans
RegionXOR
SaveRegion
```



```
Console Window
Number of points in region intersect = 81
Maximum point elevation in region intersect = 2478
Map x-coordinate of maximum elevation point = 522806.6166251945
Map y-coordinate of maximum elevation point = 1425041.0612069929
```

Database Objects

STEPS

- ☑ open the sample script DATABASE.SML from the LITEDATA / SML folder
- ☑ run the script using object HSOILS from the HAYWISOIL Project File in the LITEDATA / SF_DATA folder for input
- ☑ open the sample script DB2.SML from the LITEDATA / SML folder
- ☑ run the script using object CB_SOILSLITE from the CB_SOILS Project File in the CB_DATA folder for input

```
DatabaseGetTableInfo
FieldGetInfoByName
FieldGetInfoByNumber
NumRecords
OpenCADDatabase
OpenDatabase
OpenRasterDatabase
OpenTIFFDatabase
OpenVectorLineDatabase
OpenVectorPointDatabase
OpenVectorPolyDatabase
RecordDelete
TableAddField
TableAddFieldFloat
TableAddFieldInteger
TableAddFieldString
TableCopyToDBASE
TableCreate
TableExists
TableGetInfo
TableInsertFieldFloat
TableInsertFieldInteger
TableInsertFieldString
TableKeyFieldLookup
TableLinkDBASE
TableNewRecord
TableOpen
TableReadAttachment
TableReadFieldNum
TableReadFieldStr
TableWriteAttachment
TableWriteRecord
```

Sample script DATABASE.SML shows how to read attribute values from a database. The syntax is an extension of the TABLENAME.FIELDNAME construction used in queries. In an SML script, the database field reference must also specify the object, the database subobject (a separate database is maintained for each type of element in a vector or TIN object), and the element number. If the field being read is a string field, you must also append the "\$" character to the end of the field reference:

```
string$ = Vect.poly[4].table.field$.
```

Functions to create and modify databases are found in the Database function group. This group includes functions to create new tables, to add or insert fields in tables, to write new records in a table, and to attach records to elements in the spatial object. Sample script DB2.SML provides examples of these operations. It creates a new vector object with points located at the centroids of polygons in the input vector object, creates a point database and table, and copies selected attributes from each polygon to the associated point element.

```
# Database.sml
# sample script: Getting Started

PopupMenu("Select /litedata/sf_data/HAYWISOIL/hsouils");
GetInputVector(V);
numpolys = NumVectorPolys(V);

for i = 1 to numpolys
  acres = V.poly[i].SoilType.acres;
  type$ = V.poly[i].Wildlife.SoilName$;
  print("Polygon # %d: Soil type = %s, acres = %d\n",i,type$,acres);
}

# hsoils / PolyData / SoilType
```

DATABASE.SML refers to the ACRES field of the SOILTYPE table and the SOILNAME field of the WILDLIFE table.

Style	HapSymbol	SoilName	Acres	Percent
		107 Clear Lake clay, 0 to 2 percent slopes	8140.0	5.6
		108 Clear Lake clay, 2 to 9 percent slopes, drainage	1710.0	1.2
		109 Clinara clay, 30 to 50 percent slopes	370.0	0.3
		111 Danville silty clay loam, 0 to 2 percent slopes	10660.0	7.4

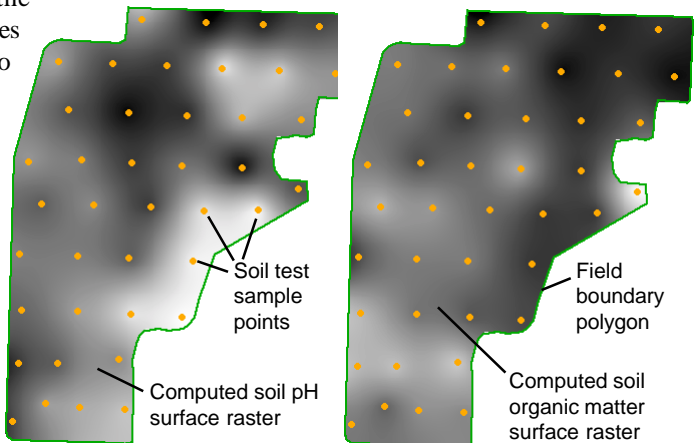
Style	HapSymbol	SoilName	Grain_Seed
		107 Clear Lake	Fair
		108 Clear Lake	Good
		109 Clinara	Poor
		111 Danville	Good

```
Console Window
Polygon # 99: Soil type = Xerorthents, acres = 3135
Polygon # 100: Soil type = Botella, acres = 4625
Polygon # 101: Soil type = Los Osos, acres = 305
Polygon # 102: Soil type = Gaviota, acres = 215
Polygon # 103: Soil type = Los Osos, acres = 1675
Polygon # 104: Soil type = Gaviota, acres = 215
```


Converting Objects

One common rationale for creating an SML script is the desire to automate a multi-step processing sequence that needs to be performed repetitively on a number of different input datasets. The ability to convert geospatial data from one type to another within SML gives you great flexibility in designing such a script. The standard TNTmips data conversion processes lead the industry in support for data types and functionality. Many of these conversion processes are available as functions in SML in the Object Conversion function group. Other specialized conversion functions in the Surface Fitting group interpolate a raster surface from a vector or TIN input object.

The SOILTEST.SML sample script automates the processing of soil sample data and uses several types of object conversion functions. The script reads a series of soil chemistry values stored in a database table attached to input vector point elements representing sample locations. For each type of value (soil pH, organic matter content, and others) the script uses a surface fitting function to create a surface raster. In intermediate steps the script uses a vector polygon representing the field boundary to create a blank raster to use as a mask for each surface. It also creates a region from the polygon and uses the region to write the value 1 into every cell in the mask raster that lies inside the field boundary.



STEPS

- open the sample script SOILTEST.SML from LITEDATA / SML
- study the script, then run it using objects in the SOILTEST Project File in LITEDATA / SML for input. Use object SAMPPTS for the "Points" and object BOUNDARY for "Boundary"
- accept the default values for the other parameters requested by popup dialog windows

```

BinaryRasterToRegion
ConvertCHYKtoRGB
ConvertHBSToRGB
ConvertHISToRGB
ConvertHSVtoRGB
ConvertRegionToVect
ConvertRGBtoHBS
ConvertRGBtoHIS
ConvertRGBtoHSV
ConvertVectorPolysToRegion
ConvertVectorPolyToRegion
ConvertVectToRegion
RasterCompositeToRGB
RasterRGBToComposite
RasterToCADBound
RasterToCADLine
RasterToTINIIterative
RasterToVectorBound
RasterToVectorContour
RasterToVectorLine
TINToRaster
TINToVectorContour
VectorElementToRaster
VectorToBufferZone
  
```

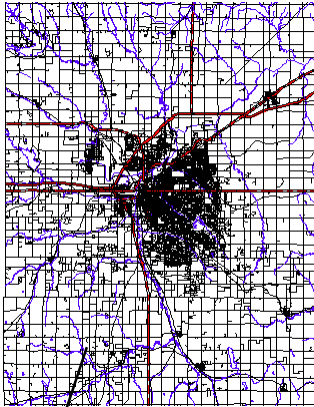
Sample Script: Extract Polygons

STEPS

- choose File / Open / *.SML File and select from your main TNT directory CUSTOM / VECTOR / TIGER.SML
- study the script structure and comments

The sample script TIGER.SML provides an example of vector and database processing in SML. It extracts specified lines from input vector objects, writes them into an output vector object, and transfers input line attributes to output polygon attributes.

TIGER.SML was designed to process vector objects imported from TIGER line files (2000 version) produced by the United States Census Bureau. TIGER geodata is organized by county, and integrates line geodata of many types (hydrology, roads, administrative and census boundary lines) into one vector data layer. Topological polygons result from the intersection of these various line types, but individual polygons have little geographic meaning. Area attributes are coded only as attributes of the left and right sides of lines. This characteristic of TIGER data makes it difficult to access and display areal information using the raw vector objects.



TIGER vector for a single county with lines styled based on their attributes.

Area boundary lines in the TIGER vector, such as city and town boundaries, can be identified by the inequality of particular attribute values on either side of the line. This script finds city boundary lines in one or more input TIGER vector objects and writes each line to a new output vector object. When all line elements for a particular city boundary have been transferred, they intersect to form a polygon in the output vector. If the current line completes a new polygon, the city name is read from the input line database, and a new polygon database record containing the name is created for the output vector. This script has been used at MicroImages to process all of the 93 county TIGER vector objects for the state of Nebraska to produce a single statewide city polygon object.



Extracted city polygons for the same county, with labels.

More about the extract polygon script is available in an online document at <http://www.microimages.com/relnotes/v65/smltiger.pdf>

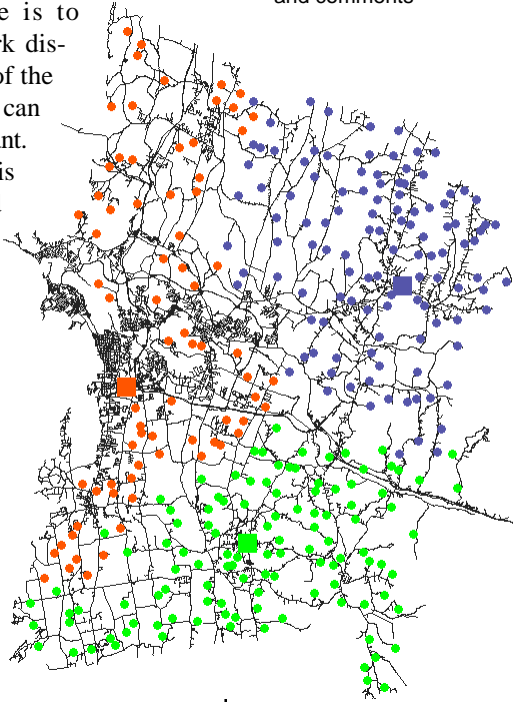
Sample Script: Network Routing

The sample script NETWORK.SML shows a more complex application of vector and database processing in SML. It uses network analysis functions to address the problem of efficient delivery of materials from numerous dispersed locations (such as farms) to a small number of destinations (such as processing plants). The objective is to determine the shortest network distance from each farm to each of the processing plants, so each farm can transport goods to the nearest plant. A script is required to solve this problem because the farm and plant locations are represented as points in vector objects separate from the object containing the road network.

For each farm and processing plant, the script adds a node to the roads object at the closest point on the closest line. It keeps track of the element numbers of these two sets of added nodes in a pair of arrays so that network distances can be associated with the correct farm and plant. Network analysis functions are then used to compute the required set of distances, which are stored in a new database table for the vector points representing farms. For each farm point, there is one attached record for each processing plant, showing the minimum network distance.

STEPS

- choose File / Open / *.SML File and select from your main TNT directory CUSTOM / VECTOR / NETWORK.SML
- study the script structure and comments



Sample result from the network script. Farm locations (circles) have been styled in the same color as the processing plant location (squares) that is closest to it along the road network.

More about the network script is available in an online document at
<http://www.microimages.com/relnotes/v65/smlnz.pdf>

Including Scripts and Running Programs

STEPS

- clear the SML window with File / New
- select Insert / Operator
- scroll to the bottom of the list in the Insert Operator window to see the SML preprocessor directives

The SML preprocessor directives can be inserted using the Insert Operator window:

```
$ifdef
$ifndef
$else
$endif
$define
$include
```

When you use the run() function as shown to the left, SML waits until you close the external program before it goes on to the next statement in the script. If you set the run() function's optional "wait" argument value to 0, the external program runs in the background.

To find out the name of a TNT process module (such as "convobjs cadtovec" in the example to the left), use any text editor to open the TNTMIPS.MNU file (which is in your TNT directory).

The SML process includes a set of preprocessor directives that are interpreted before all of the regular script statements. Preprocessor directives allow you to call up other scripts and to set up alternative script modes.

You can have a script read and execute another SML script by using the \$include directive:

```
$include "another.sml"
```

The included script should be in the same directory or Project File as the parent script. If you have several scripts that need to use the same user-defined function, the function definition can be in a separate script that you "\$include" in the other scripts.

While you are developing a complex script you might want to have a "normal" mode of execution and a "debug" mode that prints relevant information to the console to help you identify possible points of failure. You can set up the debugging mode using the directive

```
$define DEBUG
```

and bracket all of your sets of debug statements with the following pair of directives:

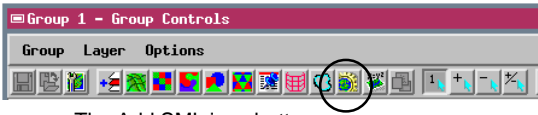
```
$ifdef DEBUG
[series of print statements]
$endif
```

To run the script in the normal mode you would simply comment out the single \$define statement, leaving your debugging code in place for later use.

If your script requires manipulations and conversions that are not supported in SML, you can use the run() system function to call TNT processes or external programs. For example, the current version of SML does not include a function to convert a CAD object to a vector object, so you might choose to have your script run Prepare / Convert / CAD to Vector:

```
run ("c:/tnt/win32/convobjs cadtovec").
```

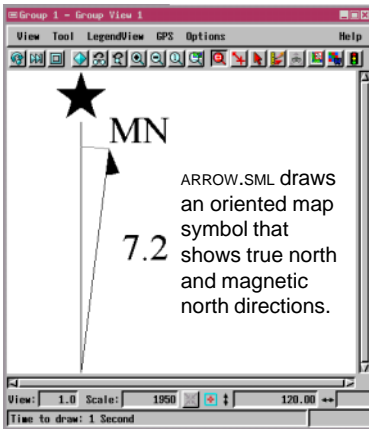
SML Layer in Display



The Add SML icon button

The standard display process (Display / Spatial Data) supports the use of an SML script as a layer, just as a raster, vector, CAD, or TIN object can be a layer. An SML script layer can use flexible cartographic drawing functions to create special map symbols and neatlines.

The sample script `ARROW.SML` is designed to draw an oriented magnetic declination map symbol in a layout. The SML layer should be alone in a group. It determines the true north direction from the previous map group in the layout. Sample script `neatline.sml` draws a neatline around a group, and includes additional drawn items that you can turn on by removing the








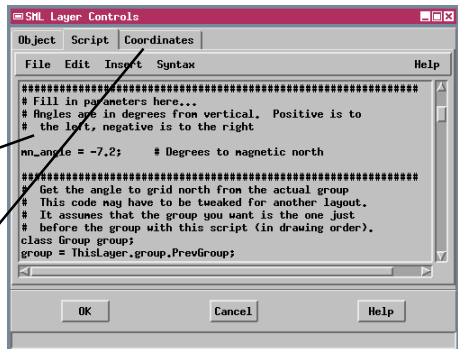
The Script tabbed panel in the SML Layer Controls window contains the interface for editing and running scripts.

The Coordinates panel lets you relate the script layer to the map coordinates of the other layers in the display.

comment character (#) from the relevant script statements.

STEPS


- run Display / Spatial Data and open a New 2D Group 
- click Add SML 
- select the Script tab in the SML Layer Controls window and choose File / Open / * .SML
- select LITEDATA / SML / ARROW.SML
- in the Coordinates panel, use the Projection button to change the coordinate system to Universal Transverse Mercator
- click [OK] to close the Layer Controls window
- examine the display, then 
- remove the SML layer
- add object _8_BIT from the CB_COMP Project File in LITEDATA / CB_DATA 
- click Add SML and select LITEDATA / SML / NEATLINE.SML 
- in the Coordinates panel, set the coordinate system to United States State Plane 1927 and the Zone to Nebraska North
- click [OK] to close the Layer Controls window



SML and GeoFormulas

A separate Getting Started booklet is dedicated to the topic of GeoFormulas. See *Getting Started: Using Geospatial Formulas*.

STEPS

- choose Display / Spatial Data
- click Add Geoformula / Quick-Add Geoformula 
- select LITEDATA / GEOFRMLA / BROV_UMN.GSF
- for input, select three TM bands from the CB_TM Project File and the SPOT_PAN image in the CB_SPOT Project File, both in LITEDATA / CB_DATA

LITEDATA / GEOFRMLA / BROV_UMN.GSF illustrates the dynamic enhancement of low-resolution TM imagery with a high-resolution SPOT image.

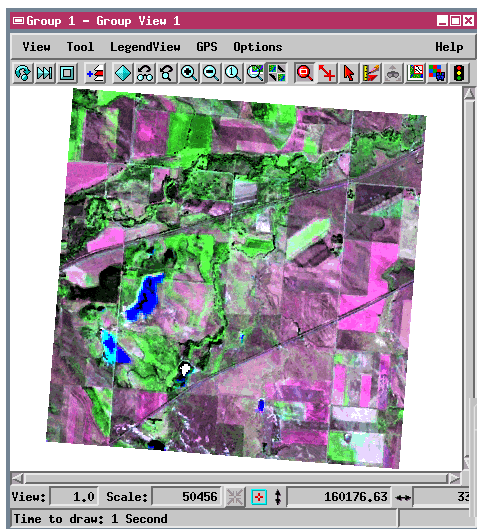
A GeoFormula layer is a computed display layer that uses one or more input objects to derive a result for display. It gives you a way to apply SML manipulations to objects “on the fly” rather than running separate processes to prepare output objects for display. A GeoFormula layer contains a “virtual object”; it does not create an output object that is saved in a Project File. Instead, it creates a display layer that releases all its system resources (such as disk space and memory) when you are finished with it.

For example, red and infrared bands of raster imagery can be combined to produce a Transformed Vegetation Index (TVI). Of course TNTmips offers a simple process that produces a TVI output raster object from selected input objects if you want to retain the TVI output for other uses. But if you just want to view the TVI result and do not care to keep the output object, you should use a GeoFormula display layer.

A GeoFormula script can be saved as a reusable file. A GeoFormula layer can be combined with any number of other layers in the TNT display process to

create a complex visualization of multiple geospatial objects.

The GeoFormula feature is primarily provided for dynamic visualization tasks in the display process. You can also run a separate GeoFormula process (Interpret / Raster / Combine / GeoFormula) to create permanent output objects for other uses.



Objects	Values	Script	Output	Preview
		$\text{sum} = \text{TM5_Value} + \text{TM4_Value} + \text{TM2_Value}$ $\text{Output_Red} = \text{TM5_Value} / \text{sum} * \text{SPOT_Value} * 3$ $\text{Output_Green} = \text{TM4_Value} / \text{sum} * \text{SPOT_Value} * 3$ $\text{Output_Blue} = \text{TM2_Value} / \text{sum} * \text{SPOT_Value} * 3$		

Creating a Simple Dialog Window

For complex scripts that include multiple operations requiring user input, consider creating a custom dialog window to control user interaction with the script. The SML functions in the Widget function group provide access to the Motif widget set, which is used to create all of the windows in the X Windows versions of the TNT products.

A dialog window consists of a parent widget that contains other component widgets. Each



widget type is a separate class in X and in SML. Sample script `DIALOG1.SML` creates and opens a very simple dialog window that displays a label string and has a Close button. Each of these components is a separate widget contained in an `XmForm` widget. An `XmForm` widget lets you place its "children" (contained widgets) using a simple relative positioning scheme. Each widget can be attached to another widget on its top, bottom, left, and right, and you can specify an offset value (in screen pixels) for each side as well. In this example the label widget (class `XmLabel`) is attached to the form on its top, left, and right sides. The Close Button (class `XmPushButton`) is attached at its top to the label widget and on the left and right sides to the form. The form widget automatically resizes to accommodate all of the contained widgets.

You can create a scrolled window using the `XmScrolledWindow` container widget in place of `XmForm` or organize child widgets into a grid using an `XmRowColumn` container widget.

STEPS

- select Process / SML / Edit Script
- choose File / Open / *.SML File and select `DIALOG1.SML` from the `LITEDATA / SML` folder
- run the script
- study the script sections that define the different parts of the Hello World dialog window
- press [Close] on the Hello World window

```
# DIALOG1.SML
# Sample script for Getting Started.
# Creates and opens a simple dialog window.

# Define parent widget for dialog window.
class XmForm win1;

# Procedure for closing window
proc OnClose() {
    DialogClose(win1);
    DestroyWidget(win1);
}

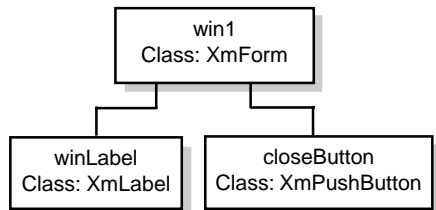
# Set up dialog window
win1 = CreateFormDialog("Hello World");
win1.MarginHeight = 5;
win1.MarginWidth = 5;

# Create label text for window
class XmLabel winLabel;
winLabel = CreateLabel(win1, "Sample Dialog Window");
winLabel.TopMidget = win1;
winLabel.LeftMidget = win1;
winLabel.LeftOffset = 10;
winLabel.RightMidget = win1;
winLabel.RightOffset = 10;

# Create Close button attached to label on
# on top and to window margin on left and right
class XmPushButton closeButton;
closeButton = CreatePushButton(win1, "Close");
closeButton.TopMidget = winLabel;
closeButton.TopOffset = 5;
closeButton.LeftMidget = win1;
closeButton.RightMidget = win1;
closeButton.BottomMidget = win1;
WidgetAddCallback(closeButton, ActivateCallback, OnClose);

# Open dialog window and keep script active
# until window is closed.
DialogOpen(win1);
DialogWaitForClose(win1);
```

Widget hierarchy in Hello World window



Using Widgets To Build Dialog Windows

STEPS

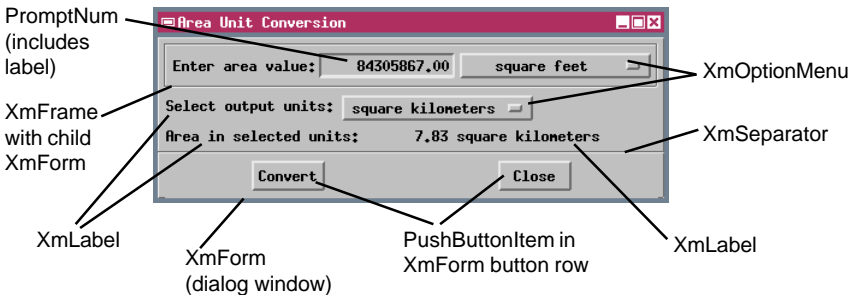
- ☑ choose File / Open / *.SML File and select DIALOG2.SML from the LITEDATA / SML folder
- ☑ run the script
- ☑ in the dialog window opened by the script, enter a value in the Enter Area Value field
- ☑ choose an input area unit from the upper unit menu
- ☑ choose an output area unit from the lower unit menu
- ☑ Press the Convert button
- ☑ study the script sections that define the different window components and actions
- ☑ click [Close] when you are finished working with the dialog window

NOTE: Script DIALOG2.SML was written to use a wide variety of widget types, not to provide an example of good window design or efficient processing. A more efficient design would omit the Convert button and recalculate the output value when any of the user settings changed.

The role of the Close button in the DIALOG1 script is defined by registering a *callback* with the widget using the `WidgetAddCallback()` function. A callback serves as a pointer to a function or procedure that a widget calls in response to one or more events. An `XmPushButton` has an `ActivateCallback` class member available to register the callback to be activated when the button is pushed. In this example, activating the Close button calls the `OnClose` procedure defined at the beginning of the script.

Sample script DIALOG2.SML creates a more complex dialog window that uses a variety of additional widget types, including a field for entering a numeric value, a frame, a separator line, and two option menus. The buttons at the bottom of the window use a different widget class than the button in the previous script. They are instances of class `PushButtonItem`, which can be used for either text buttons or icon buttons. Text buttons must be placed in a button row, a specific type of `XmForm`, and icon buttons must be placed in an `XmRowColumn` widget. You don't need to use the `WidgetAddCallback()` function to define the action of a `PushButtonItem`; the function that defines the item requires the name of the callback function or procedure as one of its arguments. The unit option menu widget also uses the latter method to define the procedure called when the unit is changed.

Widget classes used to create the Area Unit Conversion dialog window



Creating and Using a Drawing Area

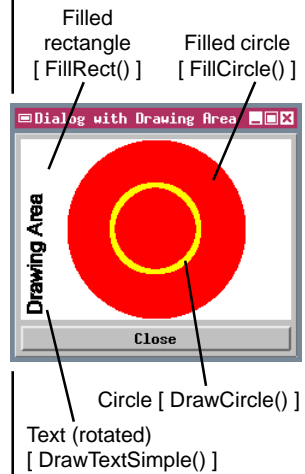
In some instances you may want to design a dialog window that incorporates a graph created from your input data or from the process output, or some other graphic. Numerous functions in the Drawing function group allow you to draw lines, geometric shapes, and text, and to set color and other style characteristics. To utilize these functions you must include an `XmDrawingArea` widget in your dialog window.

The `DIALOG3.SML` script illustrates how to set up and use a drawing area in a dialog window. When you create the drawing area, you specify its height and width in screen pixels along with the parent widget and attachment settings. Placement of elements in the drawing area is referenced to an X-Y coordinate system with units of screen pixels and an origin (0,0 position) at the upper left corner of the drawing area. When you use functions such as `SetColor()`, `SetLineWidth()`, and `DrawTextSetFont()`, these settings are used by subsequent drawing functions until you call the relevant "Set" function again to change the setting. These settings are stored in a structure called a graphics context, which is created by the function `CreateGCForDrawingArea()`. The GC must also be activated by the `ActivateGC()` function before it can be used.

If your dialog window is covered by another window and then exposed again, regular Xm widgets are redrawn automatically. If you use a drawing area, however, your script must explicitly handle this event. You must add an `ExposeCallback` to the callback list of your drawing area widget. This callback is triggered automatically when the window is opened or otherwise exposed. All of the drawing instructions must be placed inside the callback procedure so that drawing is triggered by any expose event. A graphics context requires an active window, so the GC must also be created and activated within the callback.

STEPS

- choose File / Open / *SML File and select `DIALOG3.SML` from the `LITEDATA / SML` folder
- run the script
- study the script sections that define the different window components and actions
- click [Close] when you are finished working with the dialog window



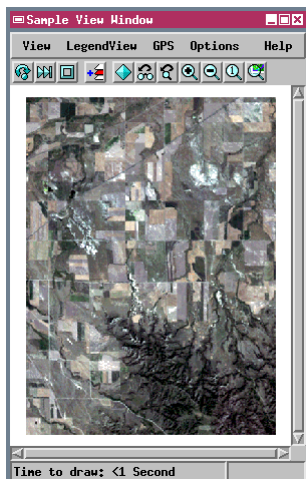
The `DIALOG3.SML` script uses a drawing area widget to draw (in order) a filled white rectangle, a filled red circle, a yellow circle, and a simple text string. The script for the Raster Profile Tool Script, described on a later page, includes a more complex example of the use of a drawing area.

Creating a View in a Dialog Window

STEPS

- select File / Open / *.SML File and open LITEDATA / SML / VIEW.SML
- run the script using as input raster _8_BIT from the CB_COMP Project File in LITEDATA / CB_DATA
- select View / Close to close the window

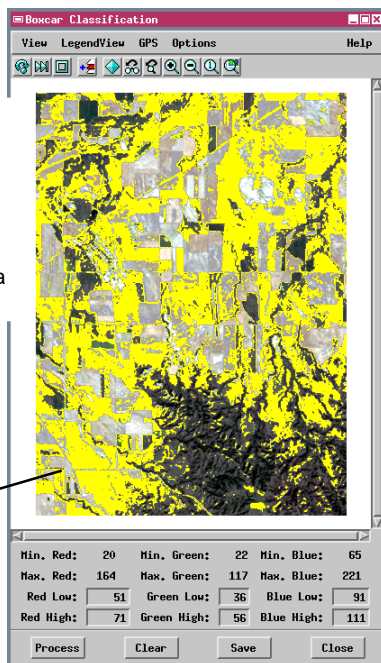
A dialog window created by an SML script can display input or output objects in a view. The GroupCreateView() function is used to create the view widget to display a geodata group within the parent dialog. Other functions in the Geodata Display, Geodata Display Group, Geodata Display Layout, and Geodata Display View function groups allow you to set up a group to display, to add objects, and to access coordinate and scale information.



Sample script VIEW.SML shows the basic steps required to open a view window of a group and display an input raster. Sample script BOXCAR2.SML creates a more complex dialog window incorporating a number of other widgets in addition to the view.

- select File / Open / *.SML File and open LITEDATA / SML / BOXCAR2.SML
- run the script, selecting for input rasters RED, GREEN, and BLUE from the CB_TM Project File in LITEDATA / CB_DATA
- press [Process] on the Boxcar Classification window to run using the default values
- study the script to see how the various window components are constructed and how actions are controlled

Sample script BOXCAR2.SML provides a more complex example of a dialog window incorporating a view.



By default, a view widget includes the standard menus, basic toolbar, scale / position line, and status line. A createflag\$ parameter of the GroupCreateView() function allows you to eliminate selected window elements if you wish. For example, the Boxcar view does not have a Scale / Position line or status line.

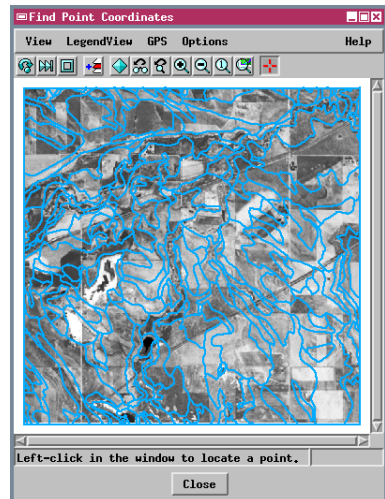
Coordinate Systems in Views

Previous exercises have discussed SML functions that use an object's georeference information to convert position information between object coordinates (such as raster line and column numbers) and map coordinates. When you display spatial objects in a view within a dialog window, several other coordinate systems come into play. Sample script PTCOORD.SML will help you explore these coordinate systems and illustrates the resources available to convert between them. The script displays a preset raster (with UTM coordinates) and vector object (with latitude/longitude coordinates) and provides a point graphic tool with which you can select a position. When you apply the tool (right-click), the point position is reported in the console window in various coordinate systems.

STEPS

- select File / Open / *.SML File and choose /LITEDATA / SML / PTCOORD.SML
- run the script
- left-click in the window to place the point tool
- right-click to view coordinates in the Console window
- try various point locations to see how the different coordinate types vary
- study the script to see how the coordinate transformations are performed
- Close the Find Point Coordinates window when you are finished

A graphic tool used in a view returns positions in *view coordinates*. For a single group view, view coordinates are the group map coordinates. The group coordinate system is determined initially by the georeference of the first layer added to the group, but can be modified by a script by resetting the Projection class for the group. *Screen coordinates* are the coordinates of the drawing area of the view (in pixels), where the objects are actually displayed. If you want the script to draw additional features into this drawing area, the drawing functions require screen coordinates. Each layer in the view also has *layer coordinates*, which are the object coordinates for the object in the layer, as well as *layer map coordinates*. The Geodata Display View function group includes functions to translate between view coordinates and screen, layer, and layer map coordinates.



```
View coordinates: x = 633864.05, y = 4730504.92
Screen coordinates: x = 67, y = 266
Raster layer (object) coordinates: x = 82.68, y = 366.80
Vector layer (object) coordinates: x = 326.33, y = 2408.09
Raster layer map coordinates: x = 633886.07, y = 4730470.08
    in Universal Transverse Mercator Zone 13 (W 108 to W 102)
Vector layer map coordinates: x = -103.36, y = 42.72
    in Latitude / Longitude
Group coordinates = View coordinates for group view.
Group coordinate system = Universal Transverse Mercator
```

Movie Generation Scripts

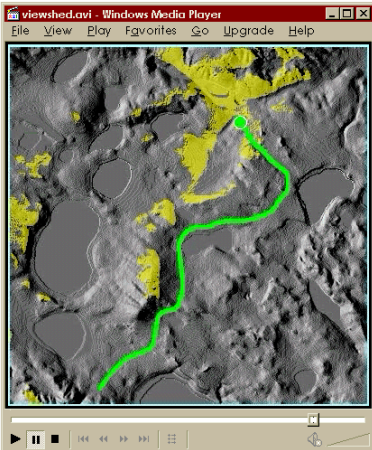
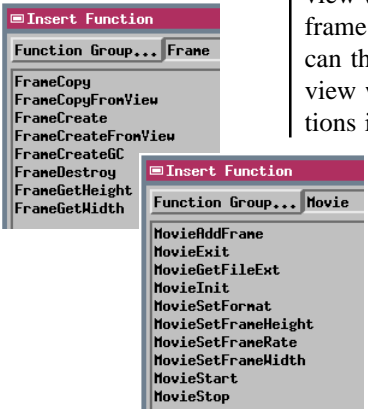
STEPS

- select Process / SML / Edit Script from the TNTmips main menu
- choose File / Open / *.SML File and select from your main TNT directory CUSTOM / MOVIE / VSHEDMOV.SML
- study the script structure and comments

An SML script can create and record custom animations from your geospatial data. The sample script in this exercise creates a movie file showing a series of viewsheds computed from an elevation raster at different points along a vector line.

Any animation consists of a gradually-varying sequence of static frames. A movie generation script captures frames from the contents of one or more view windows created by the script and copies each frame into an output MPEG or AVI file. The movie can therefore record any sequential change in the view window(s) used to create the frames. Functions in the Frame and Movie function groups are used to set up the generic frame and movie parameters, capture the view window contents to a frame, and copy the frame contents to the output file. You can also annotate each frame with text or position markers using functions in the Drawing function group.

Sequential changes in the View window can be achieved in several ways. The script could add and remove a series of pre-prepared layers to and from the view. It could also modify the display parameters for a single continuing layer. For vector objects, this could involve basing the element styles on a sequence of varying attribute values (such as population in different years). The final method is exemplified by the VSHEDMOV script: the script itself computes the changes from the supplied data and parameters. For each frame in this movie, the script computes the current viewshed and displays it in yellow over a shaded-relief rendering of the elevation model.



More about the movie generation scripts is available in an online document at <http://www.microimages.com/relnotes/v64/viewmarks.pdf>

3D Simulation Scripts

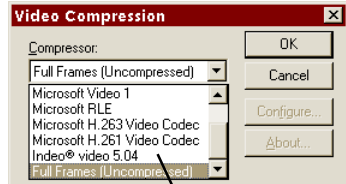
An SML movie script can also use the 3D perspective rendering capabilities of TNTmips to record custom 3D animations. A script can open a 3D perspective view window and change the viewing parameters for each frame in the movie, allowing you to move over, on, and around a 3D surface. SML incorporates all of the functionality of the 3D Simulation process in TNTmips, but expands your control over the viewing parameters.

Class members and methods in the VIEWPOINT3D class are used to manipulate the settings for the 3D view. Each 3D view has a viewer position and a position that the viewer is looking at, the point where the current view is centered. SML gives you complete control over both positions. You can set viewer and view center position coordinates explicitly for each frame, or move either position a specified distance or direction relative to the previous position. Either position can be rotated around the other. You can also set either position and then specify an azimuth angle, elevation angle, and distance to define the other.

The PATHCHT1 script copies both 3D and 2D views into each movie frame. The viewer and view center positions are computed from 2D vector lines that are displayed in the 2D view but hidden in the 3D view. The current viewer and view center positions are shown by symbols drawn into the 2D portion of each frame after the views are captured.

STEPS

- choose File / Open / *.SML File and select from your main TNT directory CUSTOM / MOVIE / PATHCHT1.SML
- study the script structure and comments



To record a movie from an SML script, you must have software capable of encoding MPEG files (any computer platform) or AVI files (Windows platform only). When recording begins, a window opens to allow you to select compression options.



Movies created from these sample SML movie scripts can be downloaded from
<http://www.microimages.com/promo/smlmovies>

APPLIDATs

STEPS

- ☑ select Custom / APPLIDAT / BENCHMRK
- ☑ click the Instructions icon button on the toolbar
- ☑ press [Close] on the Help window
- ☑ click the TNT Benchmark icon button
- ☑ try some of the benchmark processes, then press [Exit]
- ☑ click the Exit button on the toolbar

Benchmark APPLIDAT toolbar

TNT Benchmark SML script



Instructions TNTview Exit

- ☑ select File / Open / RVC Object in the SML window
- ☑ select /LITEDATA / SML / SMLAYER.RVC / ARROW
- ☑ select File / Edit Toolbar Icon
- ☑ in the Select Bitmap Pattern window, click the Set button and choose the Advisor set from the list
- ☑ select the "gold" icon illustrated and click OK
- ☑ click [Yes] to confirm your choice in the Verify dialog box



You can use SML to create self-contained, turnkey geospatial application products call APPLIDATs. An APPLIDAT can include an SML script or a series of scripts along with the geospatial data to be processed. Since data and scripts are bundled, they are loaded together automatically when the APPLIDAT is run. There is no need for the user to navigate and load the data manually. An APPLIDAT is therefore ideal for providing data with custom processing applications to users who are not familiar with the TNT interface.

An APPLIDAT includes one or more SML script objects in a TNT Project File that has been renamed with the .SML file extension. Users can run an APPLIDAT by double-clicking on the file or by using a desktop shortcut. (As shown by this exercise, TNTmips users can also run an APPLIDAT from the Custom / APPLIDAT menu.) Running an APPLIDAT launches TNTview (with the standard interface hidden) and opens a custom toolbar with an icon for each included script. Icon buttons to open the standard TNTview and to Exit the APPLIDAT also are included automatically. You can write the component scripts to use data stored in the same SML Project File or in an accompanying standard Project File in the same directory.

When a script object is created in a Project File, TNT automatically assigns it a default icon subobject, which you may edit or change for a different icon. When the APPLIDAT is launched, script icon buttons are added to the toolbar from the left in alphabetical order of the script names. If your APPLIDAT includes several scripts that should be run in a defined order, name the scripts so the alphabetical order of their names follows the defined processing sequence. A script object's description is used automatically as the ToolTip for its icon button.

Providing APPLIDAT Instructions

SML lets you write APPLIDATs that have a *discoverable* interface. Your users need not be trained in (or even aware of) the TNT products. All the instructions needed can be discovered the first time the APPLIDAT is used, or easily rediscovered after a lapse of time. Simply include in your APPLIDAT a copy of the HELP.SML script from the BENCHMARK APPLIDAT. This script creates a dialog window to display HTML-formatted text and illustrations. The HTML instruction set is stored as a subobject of the HELP.SML script.

An instruction set is easy to create and maintain because you can use any editor that supports the HTML format. Thus you can write your instructions in a program such as Microsoft Word and use its Save As... option to save the file in HTML format. To associate your new help file with the APPLIDAT, edit the HELP script in the SML script editor and select Add Text Objects from the File menu. When you select your HTML file, TNT copies it to a subobject of the script.

NOTE: to open script objects in an APPLIDAT Project File (.SML file extension) in the SML editor, you must use File / Open / *.SML File.

When you select an SML file that is actually a Project File, a Select Object window opens to allow you to select a script object from within the file.

STEPS

- choose File / Open / *.SML File in the SML window
- select BENCHMARK.SML in the Select File window
- select the Help script object in the Select Object window
- examine the script structure and comments

```

#####
#
# HELP.SML
#
# A sample script that shows how to create a simple dialog with HTML
# help and a close button
#

class XnForm form;
class XnPushButton button;

# The function that will be called when the "close" button is clicked
#-----
# Create a dialog. The string passed here is the title of the dialog.
# Eventually it will pull the title out of the <title> tag in the HTML
form = CreateFormDialog("Help");
form.height = 400;
form.width = 500;

#-----
# Create a close button

# Called when the user clicks the "Close" button. Just close the dialog.
proc cbClose() {
  DialogClose(form); # Will cause a popdown, causing cbPopdown to be ca
}
  
```

You can copy this Help script to your own APPLIDAT file and use it directly to create your Instructions or Help window. An instruction set won't become separated from its APPLIDAT because it is bundled with the other resources.

BIOMASS2 APPLIDAT

Biomass Mapping

Instructions



Asset Management

3D Simulation

STEPS

- select Support / Maintenance / Project File from the TNTmips main menu and examine the contents of BIOMASS2.SML in the Custom / APPLIDAT folder in your main TNT products folder
- exit from Project File Maintenance and select Custom / APPLIDAT / BIOMASS2
- click the Instructions icon button and read the instructions
- click on the Biomass Mapping icon button, define an area to map, filter the result, and convert the result to a vector
- exit from the Biomass Mapping window
- run the Asset Mapping and 3D Simulation applications
- exit from the BIOMASS2 APPLIDAT when you are finished

The BIOMASS2 APPLIDAT was written by MicroImages to provide an example and prototype of a turnkey APPLIDAT product. It illustrates how an APPLIDAT can let the user carry out a series of operations on the input data and automatically pass intermediate products along to the next operation. In this example the application would allow a farmer to determine crop biomass for any designated area from a color infrared image, display farm assets over the image and biomass map, and display a 3D perspective view of the image and biomass map. The Instructions for the BIOMASS2 APPLIDAT provide a more detailed overview of each operation.

The APPLIDAT file (BIOMASS2.SML) includes three processing script objects: Biomass (Biomass Mapping), Pinmap (Asset Management), and View3D (3D Simulation) that are designed to be run in that order (note the alphabetical order of the script names and the positions of their icons in the toolbar). Instructions for the product are contained in the script called About (note that the script itself contains the HTML formatted instructions, rather than using an HTML subobject). All of the input data are in the APPLIDAT file. Spatial objects produced by the APPLIDAT are stored and retrieved as needed in an accompanying Project File BIOMASS.RVC.

After you have run the APPLIDAT, you should examine the structure of the component scripts. Each script contains code to create its dialog window and controls, callback procedures assigned to those controls, and instructions for input and output of data. You can use these as models in developing your own turnkey APPLIDAT programs.

Tool Scripts and Macro Scripts

Tool Scripts and Macro Scripts are specialized forms of SML scripts that are launched from an icon button in a View window and can automatically access and operate on the objects in the view. You can create tool scripts or macro scripts that enable any user to perform custom procedures on spatial data layers loaded into the view. After you add a tool script or macro script, its icon button appears on the toolbar of every View window across all TNT processes. And every View window offers menu selections that let you easily add and delete Tool scripts and Macro scripts (Options / Customize)



Tool scripts and macro scripts are launched from icon buttons on a View window's toolbar.

For the script writer, macro scripts and tool

scripts provide a streamlined way to provide custom processing capabilities that require visual interaction with the spatial data. To do this in a standard SML script, you have to provide the code to create and manage the View window and its contents. But because macro scripts and tool scripts are invoked from a View window, most of that management is taken care of automatically, and you can focus on coding the custom processing itself.

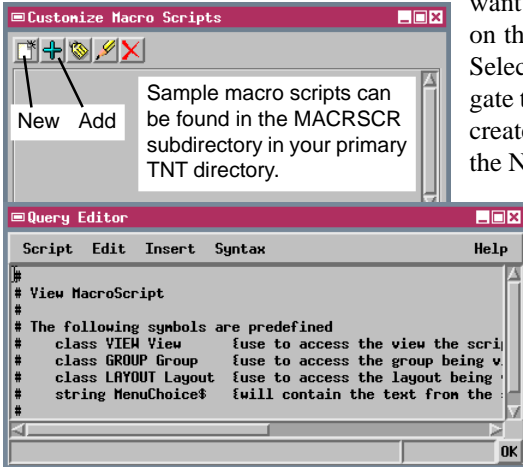
Macro scripts and tool scripts:

- are executed from an icon button on a View window toolbar;
- can access features of the current view, such as layers, extents, projection, selected elements, zoom factor, scale, and styles;
- can operate on objects in the current view or objects containing the same area;
- can add a newly-created layer to the view;
- can start an external program and provide it with data derived from the current view.

A tool script invokes a drawing tool and/or a dialog window (defined by the script-writer) that allow the user to interact with the spatial data in the view window. For example, the user could outline an area or select particular elements to be processed. A macro script does not allow such graphical interaction, but can be set up with a drop-down menu that provides program options.

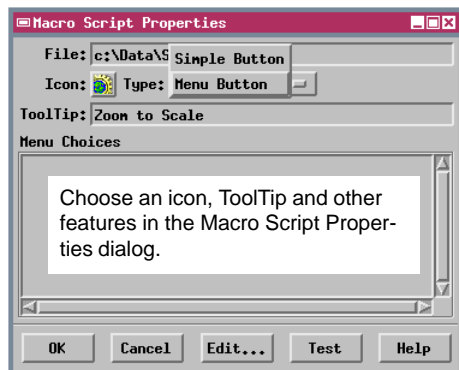
Macro Script Setup

To add a macro script so it can be run from an icon on the View window toolbar, choose Options / Customize / Macro Scripts from the View window in any process. Making this selection opens the Customize Macro Scripts window. If you



want to add an existing script, click on the Add icon button to open the Select File window so you can navigate to the script and select it. To create a new macro script, click on the New icon button. A Query Editor window opens with a default script containing a list of predefined symbols that you can use in the macro script. The Query Editor window includes all the script-creation and editing features of the standard SML window.

Once you have created or added the macro script, the Macro Script Properties window opens. This window lets you choose an icon, indicate whether the script is launched from a simple button or a menu button, set up the ToolTip for the icon button, enter menu items if a menu button is used, and test your script. Choose a simple button to have your tool script execute automatically without further input from the user. Choose a menu button if you want drop-down choices presented when the button is clicked. If Menu Button is chosen in the Macro Script Properties window, the Menu Choices text field becomes active so you can enter the menu choices needed for the script.



Enter the ToolTip you want directly in the ToolTip field. This ToolTip appears when the cursor hovers over the macro script's icon in the View window. The Test button at the bottom of the window lets you run your script without closing the customize windows. Click OK in the Macro Script Properties and Customize Macro Scripts windows when you are done adding, developing, and/or testing your script.

Click OK in the Macro Script Properties and Customize Macro Scripts windows when you are done adding, developing, and/or testing your script.

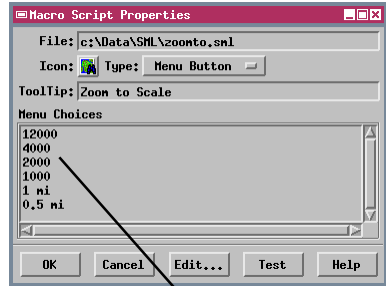
Sample Macro Script: Zoom to Scale

Several sample macro scripts are provided in the `MACRSCR` subdirectory in your primary TNT directory. Study these samples to understand how to structure your own macro scripts.

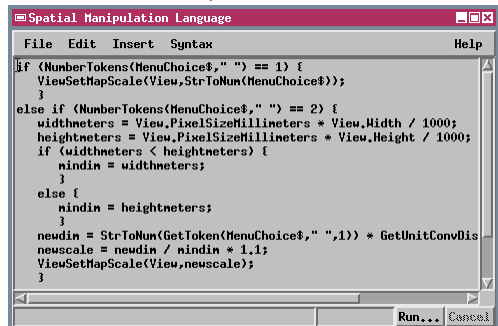
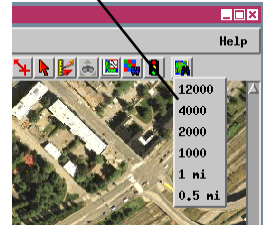
The Zoom to Scale macro script lets the viewer redisplay the View window at one of several map scales selected from the script button's dropdown menu. For proper script function, the objects in the view window must be either georeferenced or scale-calibrated.

The menu selections are not predetermined by the Zoom to Scale script. When you install the script, you are free to set up the menu choices with the range of scale selections most appropriate for your data. The script accepts scale input from the menu as either map scale or ground dimensions. If the menu entry is purely numeric, it is interpreted as the denominator of the map scale fraction. For example, 12000 is interpreted as a map scale of 1:12000. If the menu entry is in two parts separated by a space (such as "1 mi"), the first part of the entry is interpreted as a ground dimension in miles. (This portion of the script can be easily modified to accept dimensions in kilometers or other distance units.) The script then performs the necessary calculations and sets the new map scale for the View window.

The predefined macro script variable `MenuChoice$` is used to represent the user's selection from the macro script menu button. For numeric input, this string must be converted to a numeric value using the `StrToNum()` function.



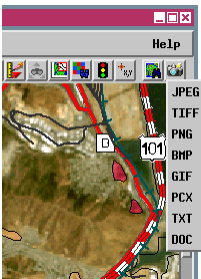
When installing the Zoom to Scale script, set up scale menu choices that are most appropriate for your spatial data.



More about the Zoom to Scale macro script is available in an online document at

<http://www.microimages.com/relnotes/v64/zoomto.pdf>

Sample Macro Script: Snapshot



The Snapshot script is a simple example of a macro script that processes data from a View window and launches an external application. The script captures a screen snapshot of the view window and exports it to the image file format you have chosen from the script button's dropdown menu. The script then launches the application program that you have previously registered with your operating system to open that file type.

```

Spatial Manipulation Language
File Edit Insert Syntax Help
# Export and open the snapshot.
if (MenuChoice$ == "JPEG") {
  OpenRaster(ras, rast.$Info.FileName, "snap");
  ConvertCompToComp(ras, rast.$Info.FileName, "snapa", 24);
  CloseRaster(ras);
  jpegHandle.exportCompressFactor = 75;
  ExportRaster(jpegHandle, _context.ScriptDir + "snapshot.jpg", r);
  RunfAssociatedApplication(_context.ScriptDir + "snapshot.jpg");
}
else if (MenuChoice$ == "PNG") {
  OpenRaster(ras, rast.$Info.FileName, "snap");
  ConvertCompToComp(ras, rast.$Info.FileName, "snapa", 24);
  CloseRaster(ras);
  ExportRaster(pngHandle, _context.ScriptDir + "snapshot.png", r);
  RunfAssociatedApplication(_context.ScriptDir + "snapshot.png");
}
else if (MenuChoice$ == "BMP") {
  OpenRaster(ras, rast.$Info.FileName, "snap");
  ConvertCompToComp(ras, rast.$Info.FileName, "snapa", 24);
  CloseRaster(ras);
  ExportRaster(bmpHandle, _context.ScriptDir + "snapshot.bmp", r);
  RunfAssociatedApplication(_context.ScriptDir + "snapshot.bmp");
}
Run... Cancel
    
```

The Snapshot script has been written to create specific file formats: JPEG, PNG, BMP, PCX, GIF, TIFF, and ASCII files with either TXT or DOC file extensions. When you add this macro script to a View window, you must set up choices for the script button menu from this set of formats. The text for each menu entry must exactly match the character string expected by the script, including case (for example, JPEG rather than Jpeg).

Saved snapshot of View window with raster background and several vector overlays.



The script initially saves the snapshot as a temporary color composite raster object. The bit depth of the composite is determined by your computer's display settings. The script segment for each file format performs a color conversion to the color depth appropriate for that format prior to export. The output file is automatically saved in the same directory as the script, then the file's associated application is launched. These operations make use of a class variable `_context`, which specifies the internal context information for the script. Class member `_context.ScriptDir` specifies the directory in which the script is found.

Sample Tool Script: Select Point

A number of sample tool scripts are provided with the TNT products in the `TOOLS.CR` subdirectory under your primary TNT directory. You can use components from any or all of these scripts to create the custom tool you need for your specialized application.

The point selection script (`POINTSEL.SML`) illustrates how to set up a tool script that lets the user interactively select elements from a vector object in the View window. In this case the script selects the closest point element when the left mouse button is pressed; this action is controlled by the definition for the `OnLeftButtonPress()` function. This simple script merely selects the point, but the button press function could be expanded to use the selected point for

further processing, such as writing the map coordinates of each point to an external file.

Because a toolscript is executed interactively from a View window, all processing is carried out by script functions executed by mouse actions or by actions carried out in dialog windows created by the script. The function definitions you provide for the predefined function names can call other functions and procedures defined elsewhere in the tool script. In the point selection script, for example, the `OnLeftButtonPress()`

```

# POINTSEL.SML - Allows user to select and highlight a vector point.
# Created by: Mark Smith
# Most recent revision: 9-2001

# The following symbols are predefined
# class VIEW View           {use to access the view the tool script is
# class GROUP Group        {use to access the group being viewed if t
# class LAYOUT Layout      {use to access the layout being viewed if t

# The following values are also predefined and are valid when the various (
# functions are called which deal with pointer and keyboard events.
# number PointerX         Pointer X coordinate within view in pixels
# number PointerY         Pointer Y coordinate within view in pixels

# Variable declarations
class VECTORLAYER vectorLayer;
class Vector targetVector;
class GROUP activeGroup;

# Checks layer to see if it is valid.
func checkLayer() {
    local boolean valid = true;

    # Get names layers if usable. If not output error messages.
    # Get name of active layer if it is usable. If not output an error mess
    if (activeGroup.ActiveLayer.Type == "") {
        PopupMessage("Group has no layers!");
        valid = false;
    }
    else if (activeGroup.ActiveLayer.Type == "Vector") {
        vectorLayer = activeGroup.ActiveLayer;
        DisplayVectorFromLayer(targetVector, vectorLayer);
        if (targetVector.Info.NumPoints < 1) {
            PopupMessage("No points!");
            valid = false;
        }
    }
    else {
        PopupMessage("Not a vector!");
        valid = false;
    }
}

```

Tool scripts can include user-defined procedures and functions that are called by other functions in the script.

function calls a previously-defined `checkLayer()` function that checks to make sure that the active group contains a layer, and that the layer is a vector object. The `OnInitialize` function also calls a procedure `cbGroup()` to identify the active group in a multigroup layout. This code generalizes the tool script for use in either a group view or layout view window.

Sample Tool Script: ViewMarks

The ViewMarks tool script (VPTOOL.SML) allows you to record a list of position markers for the View window. A ViewMark records the map coordinates of the current view center (in latitude/longitude) and the map scale. Once the list is created, you can select a ViewMark and recenter the View window on that location at the designated scale. ViewMarks are particularly useful for layouts that cover a large geographic area, especially when the layout uses limited map scale visibility to add and remove layers as you zoom in and out.



VPTOOL.SML lets you pick a viewpoint from the Viewpoint List to center the view at that location and scale.

The ViewMarks script creates a Viewpoint List dialog window that provides an interactive list as well as buttons used to initiate script actions; there is no graphic tool created by the script. This dialog is created by the OnInitialize() function. The icon buttons on the window let you add or remove ViewMarks from the list and zoom to the selected mark. Other push buttons let you save the list to a text file, open an existing viewpoint list file, create a new list, or close the window. Each of these buttons calls a separate function or procedure defined in the tool script.

```

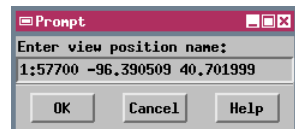
Spatial Manipulation Language
File Edit Insert Syntax Help
#
# ToolScript for recording "viewpoint position" as center and zoom.
#
# The following symbols are predefined
#   class VIEW View           fuse to access the view the tool script
#   class GROUP Group        fuse to access the group being viewed
#   class LAYOUT Layout      fuse to access the layout being viewed
#   number ToolIsActive      Will be 0 if tool is inactive or 1
#
class XmForm dialog;
class XmlList poslist;
class MAPPROJ projLation;
class TRANSPARM transMapToView;
class FILE posfile;

number ischanged;
number setDefaultWhenClose;
number numpos;
array posX[11];
array posY[11];
array posScale[11];

# Save the list to a file.
func DoSave () {
  if (numpos == 0) return;
  posfilename$ = GetDefaultFileName("", "Select position file to save");
  DeleteFile(posfilename$);
  # If you get an error that fopen() is being passed too many parameters
  # get a new tntdisp.exe. The 3rd parameter was added 01-Feb-200
  posfile = fopen(posfilename$, "w", "UTF8");
  if (posfile == 0) return (false);
}

```

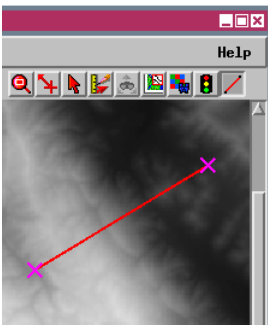
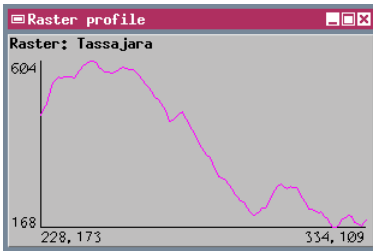
When you add a ViewMark, a prompt window opens to let you name the mark. (The default name is the zoom level and coordinate position). The ViewMark names are stored in a list widget (class XmlList). The x-coordinate, y-coordinate, and scale values are stored in separate numeric arrays.



More about the ViewMarks tool script is available in an online document at
<http://www.microimages.com/relnotes/v64/viewmarks.pdf>

Sample Tool Script: Raster Profile

The Raster Profile tool script (RASTPROF.SML) provides a line tool that records and plots a profile of the raster cell values along a line drawn by the user. The target raster for the profile must be the active layer in the view, and x-y positions for the values are recorded in raster coordinates (column and line number). Although the profile plot is the end result in this example, the script can be modified to convert positions to map coordinates, apply additional processing to the profile values, or write them out to a text file.



A portion of the OnInitialize() function in the script invokes a standard interactive line tool:

```
tool = ViewCreateLineTool(View);
ToolAddCallback(tool.ApplyCallback,
                cbToolApply);
```

(The variable `tool` was previously declared as a member of class `LineTool`.) The procedure `cbToolApply()`, which acquires the profile, is called when the tool is applied by a right-mouse-button press.

This linkage is set up by the second statement in the excerpt above, which adds the procedure name to the tool's `ApplyCallback` list. This structure dispenses with the need for a separate `OnRightButtonPush` function.

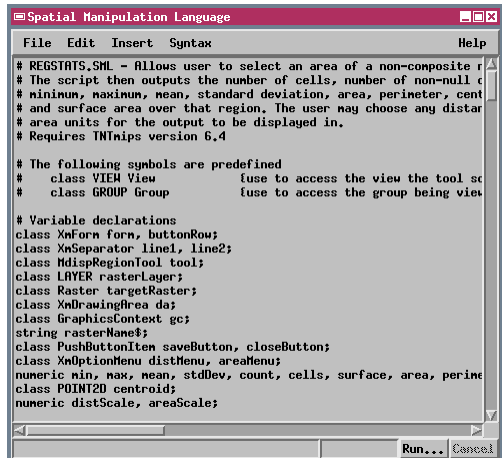
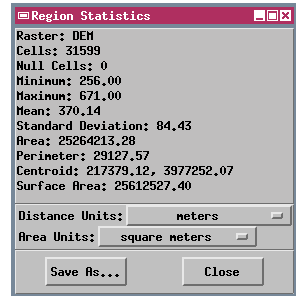
The script also demonstrates how the result of an action can be shown graphically in a window created by the script. The code that draws the graph axes, labels, and profile is contained in the procedure `cbRedraw()` defined in the script.

```
Spatial Manipulation Language
File Edit Insert Syntax Help
#####
# RASTPROF.SML
# Created by: Mark Smith
# Most recent revision: 8-2001
#
# This toolscript allows the user to draw a line using a line tool,
# the raster data from the active raster along the line and then
# the profile of the raster along the line.
#
# The active layer may not be a composite raster.
#
# This script requires TNTips version 6.6.
#
# The following symbols are predefined
# class VIEW View           {use to access the view the tool sc
# class GROUP Group        {use to access the group being vie
#
# Variable declarations
class XnForm form;
class LineTool tool;
class LAYER rasterLayer;
class Raster targetRaster;
class XnDrawingArea da;
class GraphicsContext gc;
class GROUP group;
string rasterName;
numeric doGraph, hasNull;
array value[1000000];
array draw[1000000];
array graphx[3], graphy[3];
numeric min, max, count, nullVal;
class POINT2D startpoint;
class POINT2D endpoint;
```


Sample Tool Script: Area Statistics

The Area Statistics tool script (REGSTATS.SML) shows how you can create a custom tool to let the user draw a polygon in the view window, convert the polygon to a region, and use the region to operate on another object. In this example, the region is used for the simple task of extracting statistics from a raster layer in the view. But the script could be modified to perform many other functions, such as creating a mask raster or extracting elements from a vector object. The region operations are not restricted to layers in the view; you can operate on any georeferenced objects that overlap the defined region.

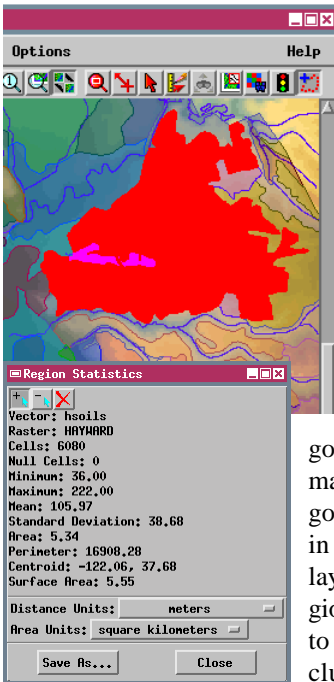
This script operates on a raster object that is the active layer in the view. In the example shown here, the polygon is drawn on an image layer overlying the active layer, which contains an elevation raster. Using the region defined by the polygon tool, the script computes the number of cells, number of null cells, minimum, maximum, mean and standard deviation of the included raster values, and the area, perimeter, centroid location, and surface area of the region. (Statistics can be computed for any type of grayscale or binary raster, but not for composite rasters or RGB raster layers.) The statistics are shown in a Region Statistics dialog window created by the script. The script can convert distance and area values to the units selected from option menus on the window. The statistics can also be saved to a text file.



More about the Area Statistics tool script is available in an online document at

<http://www.microimages.com/reInotes/v64/polystats.pdf>

Sample Tool Script: Region Statistics

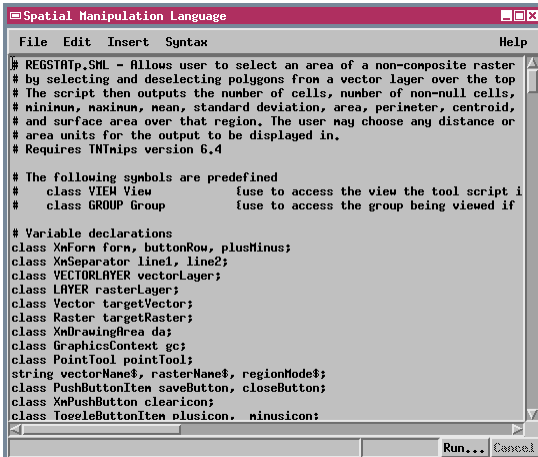


The Region Statistics tool script (REGSTATP.SML) demonstrates the design for a script that lets the user select polygons from the view window, creates a region from the selected polygons, and uses the resulting region to perform an action on another object. The example task for this script is the same as for the Area Statistics tool script: compute statistics from a raster layer in the view. Like that script, however, you could rewrite the `cbToolApply()` procedure to perform different types of operations on other objects.

This script lets you select one or more polygons from the top layer in the view (and checks to make sure that that layer is a vector object with polygons). Statistics are computed for the bottom layer in the view; the script checks to make sure that that layer is a grayscale or binary raster object. The Region Statistics window created by the script is similar to the one used by the Area Statistics script, but includes push-buttons at the top that let the user

indicate whether the selected polygon should be added to or subtracted from the region, and a button to clear the region.

The Region Statistics script invokes a standard point tool with predefined mouse button actions. A left button press places the point tool, and a right button press selects the enclosing polygon.



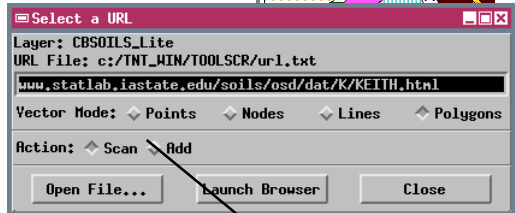
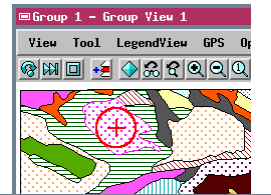
More about the Region Statistics tool script is available in an online document at
<http://www.microimages.com/relnotes/v64/regionstatistics.pdf>

Sample Tool Script: Run Browser

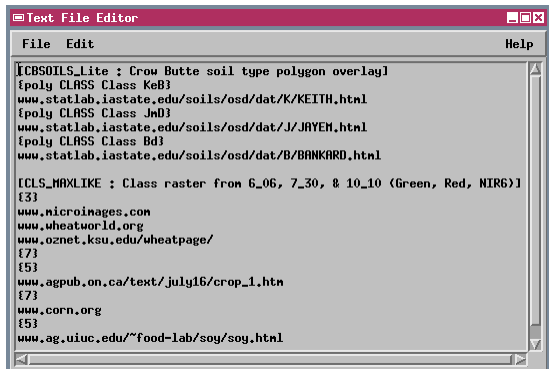
The Run Browser tool script (URLS.SML) is an example of a custom script that launches an external application program. The script allows a user to set up and use links between spatial data in a view window and sites on the World Wide Web. Links can be made to cell values in a raster, or to specific attribute values associated with vector elements. One or more URLs can be entered for each value. Once links are set up, the user can select an element or cell in the view window, choose the desired URL, then have the script launch the default web browser, which then goes to the desired web address.

To use the tool, left-click on the polygon or cell desired, then right-click to confirm the select tool is correctly positioned. The URL(s) associated with the selected feature appear in the Select a URL window, then click on the Launch Browser button.

The associations between URLs and element attributes or cell values are stored in a separate text file, specified in the sample script as URL.TXT. The text file lists the name and description for each object with URL links. The associations in this sample tool script refer specifically to CB_DATA / CB_SOILS.RVC / CBSOILS_LITE, or BERA / BERCRPCL.RVC / CLS_MAXLIKE.



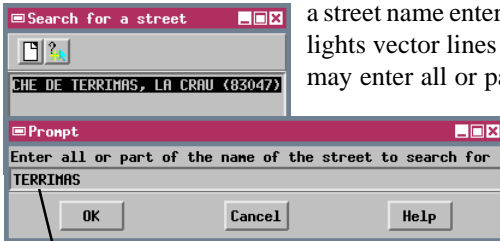
Turn on the Add button to set up links, and the Scan button to use existing links. To use links in Scan mode, select your target URL and click [Launch Browser].



More about the Run Browser tool script is available in an online document at
<http://www.microimages.com/relnotes/v64/runbrowser.pdf>

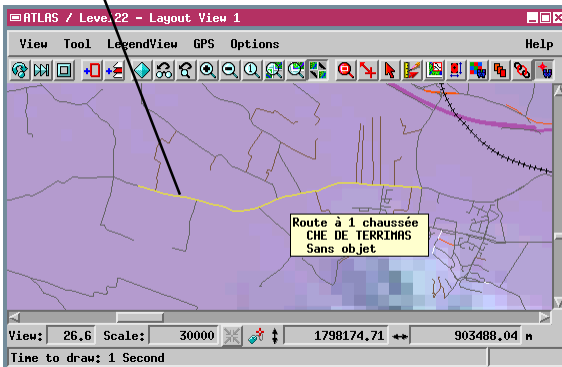
Sample Tool Script: Find Streets

The Find Streets tool script (STREETS.SML) illustrates how a script can access database information and perform specialized selection tasks. The script uses a street name entered by the user to locate and highlights vector lines representing the street. The user may enter all or part of a street name, and the tool script displays a list of all streets containing that search text. When the user picks a street from the list, the script redraws the view at 1:30000



The user enters a street name and the tool script finds it on the map.

with all lines that form parts of the street highlighted and centered in the View. If all the street's lines do not fit in the View at 1:30000, the View is redrawn at a scale that fully contains the lines.



The script uses the current highlight colors for selected and active elements (Options / Colors). For this tool, the selected street will have a uniform appearance if both the active and selected colors are the same (yellow in the window illustration).

STREETS.SML is coded to work with specific geodata from a sample atlas of France. You must modify the script before it will work with other geodata and attributes.

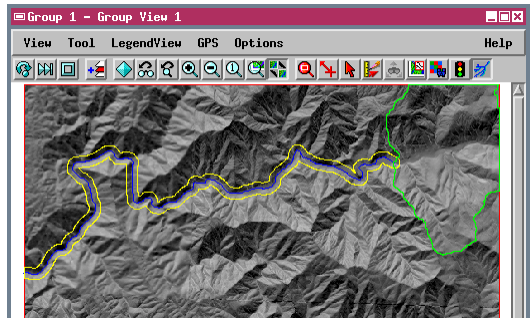
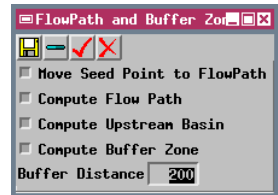
The name of the town and the zip code are also provided in the list of streets found. The script assumes there are not two separate streets in the same zip code with the same name. If, however, it turns out that the search name belongs to two different streets in the same zip code (one Main Street, the other Main Drive, for example), only the first encountered is listed but both are highlighted when that selection is made.

More about the Find Streets tool script is available in an online document at <http://www.microimages.com/relnotes/v64/findstreets.pdf>

Sample Tool Script: Flow Path

The Flow Path tool script shows how custom analysis procedures can be performed on layers in the current view using an SML Tool Script. The script uses SML watershed functions that operate on an elevation raster (DEM) that must be the first layer in the View window.

When the user launches the script, it first executes watershed functions to create a depressionless version of the DEM and a complete set of vector flow paths. These derived features are required by subsequent script steps; they are stored as temporary objects and are not displayed in the view. The script then opens a FlowPath and Buffer Zone window and creates a graphic tool that allows the user to place one or more watershed seed points on the DEM or on an overlying image layer. Toggle buttons on the window enable the user to choose to compute and display:



- the upstream basin (area with flow toward the seed point),
- the flow path downstream, and
- a buffer zone around the flow path.

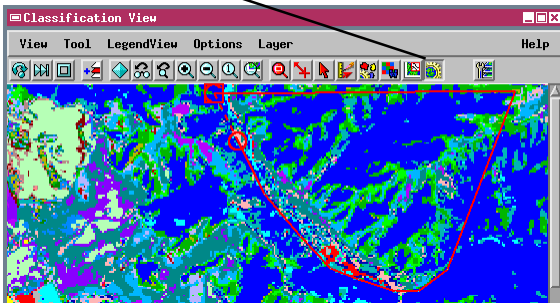
If the user intends seed points to fall along a stream course, they can turn on the Move Seed Point to Flow Path toggle button. Each seed point is then moved to the nearest precomputed flow path line before the new flow path and basin are computed. The user can place new seed points and repeat the analysis as many times as desired, and save the computed vector objects.

The script also creates and displays (in red) a vector layer outlining the extents of the DEM. If an overlying image layer is larger than the DEM, the user can use the extents box to guide placement of the seed points. The extents box is also used to automatically clip buffer zones computed from flow paths that intersect the DEM boundary.

More about the Flow Path tool script is available in an online document at
<http://www.microimages.com/relnotes/v64/flowpath.pdf>

Sample Tool Script: FRAGSTATS

Once installed, a tool script can be run from any view window. So you can run the Automatic Classification process and immediately run the Fragstats tool script on part of the Class raster that is shown in the Classification View window.



The FRAGSTATS tool script (FRAGTOOL.SML) is an example of a script that extracts spatial data from a raster layer in the view and passes the data to an external application program for processing. The FRAGSTATS program was developed by landscape ecologists to compute a variety of statistics about the spatial patterns of areas (patches) representing different ecological habitat classes. The appropriate

input for the tool is therefore a class raster, one that has a unique integer value assigned to cells of each category or class. You can create class rasters from multispectral imagery using the Automatic Classification or Feature Mapping processes in TNTmips.

A separate script for running FRAGSTATS from the SML process interface is also available. FRAGSTAT.SML can be found in your main TNT directory under / Custom / General. This script requires that you provide both the class raster and a binary mask raster to define the area of interest.

The FRAGSTATS tool script provides a polygon tool that lets the user select an area (created as a temporary region object) for calculating the landscape statistics. When the tool is applied, the script writes the class raster to a text file for use by the FRAGSTATS program. Cells outside the region of interest are given negative class values in the text file, which is the FRAGSTATS convention for identifying cells that are outside the "landscape boundary". The script then launches the FRAGSTATS program in a DOS shell. FRAGSTATS identifies homogeneous patches and computes statistics for the individual patches and for entire classes. The statistics are saved in a series of text files.

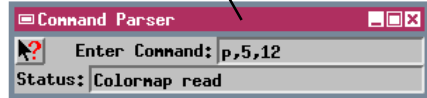
More about the Fragstats tool script is available in an online document at

<http://www.microimages.com/relnotes/v65/fragstats.pdf>

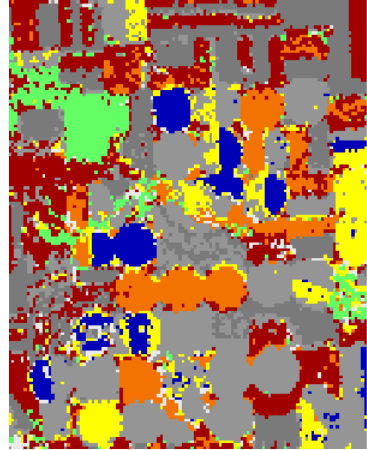
Sample Tool Script: Command Parser

Several of the tool scripts discussed previously create a control window that allows users to execute script actions using push buttons or other graphical interface controls. The Command Parser tool script (COMPAR.SML) demonstrates a script design that creates a "command line" interface for executing script actions. The Command Parser window created by the script includes a text field in which the user enters predefined text commands. A procedure named ParseCommand() associates each command string with a particular function or procedure defined elsewhere in the script.

The Command Parser window created by the script includes a field for entering command strings and one that displays process status messages. An icon button opens a Help dialog window.



This sample script was designed as a command-line equivalent to the graphical Color Palette Editor in TNTmips. It allows a user to create or edit a color palette by assigning colors to particular cell values or cell value ranges in a raster. The script uses a very small set of commands (each one or two characters long), some of which are accompanied by numeric parameters. For example, the command string "pr,3,20,1" paints a range of cell values from 3 to 20 with the color specified by color index number 1. The index numbers and corresponding color values (R, G, B, and Transparency values) are defined in a text file, which for script access must be read into an array using the command "b".



Although a graphical interface is easy to learn, experienced users can execute repetitive tasks more quickly using a command-line interface. Tasks that might require several mouse actions in a graphical window can be executed using a single short command string.

Commands are included to create a color text file from a color palette in a project file, or to create a color palette from a text file.

More about the Command Parser tool script is available in an online document at

<http://www.microimages.com/relnotes/v65/fixcolor.pdf>

Advanced Software for Geospatial Analysis

MicroImages, Inc. publishes a complete line of professional software for advanced geospatial data visualization, analysis, and publishing. Contact us or visit our web site for detailed product information.

TNTmips TNTmips is a professional system for fully integrated GIS, image analysis, CAD, TIN, desktop cartography, and geospatial database management.

TNTedit TNTedit provides interactive tools to create, georeference, and edit vector, image, CAD, TIN, and relational database project materials in a wide variety of formats.

TNTview TNTview has the same powerful display features as TNTmips and is perfect for those who do not need the technical processing and preparation features of TNTmips.

TNTatlas TNTatlas lets you publish and distribute your spatial project materials on CD-ROM at low cost. TNTatlas CDs can be used on any popular computing platform.

TNTserver TNTserver lets you publish TNTatlases on the Internet or on your intranet. Navigate through geodata atlases with your web browser and the TNTclient Java applet.

TNTlite TNTlite is a free version of TNTmips for students and professionals with small projects. You can download TNTlite from MicroImages' web site, or you can order TNTlite on CD-ROM.

Index

APPLIDAT.....	3,38-40	Macro Script.....	41-44
assignment statement.....	5	movie script.....	36,37
CAD objects.....	22	procedures.....	10
database objects.....	24,26,27	raster objects.....	19,25
dialog windows, creating.....	31-35	region objects.....	23,25
classes.....	11-13	SML layer.....	29
custom menu.....	17	TIN objects.....	22
encryption.....	18	Tool Script.....	41,45-56
expressions.....	7	toolbars.....	17
functions.....	8-10	variables.....	6
GeoFormula.....	30	vector objects.....	20,21,26,27
including scripts.....	28	widgets.....	31-35
loops (for, for each, while).....	15		



MicroImages, Inc.

11th Floor – Sharp Tower
206 South 13th Street
Lincoln, Nebraska 68508-2010 USA

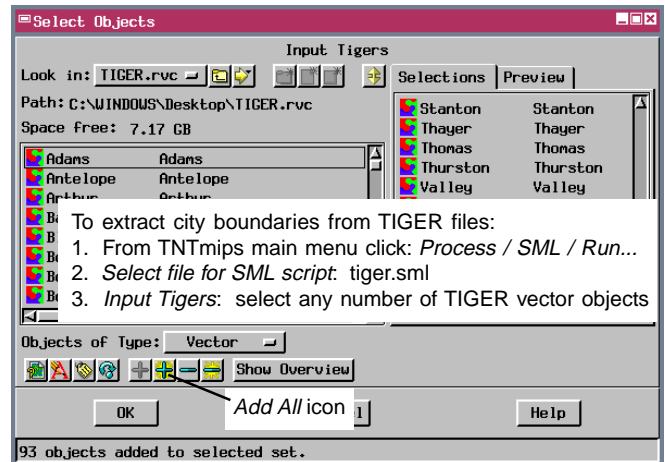
Voice: (402)477-9554
FAX: (402)477-9559

email: info@microimages.com
Internet: www.microimages.com

Sample SML Script

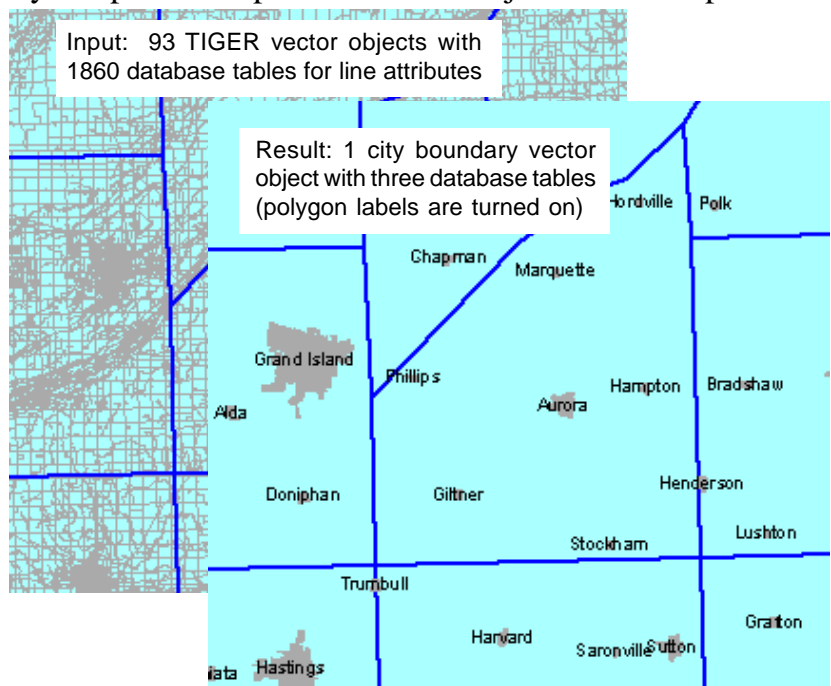
Extract Selected Polygons

The tiger.sml script extracts city boundaries from TIGER / Line files that have been imported into TNTmips. TIGER geodata is a digital map database used to support the United States Census Bureau's census and survey programs. TIGER / Line geodata has a standard vector and attribute database format that is the same for each of the 3142 counties in the USA. This script can be applied to any county TIGER file to extract and create a vector object containing only city boundary polygons and an attribute table of city names. All the extensive additional graphical and tabular data in these files is excluded from the new vector object. This much smaller, special purpose vector object requires less storage and is significantly faster when used in a display, to build an index, or to search.



This SML script demonstrates how to extract specified elements from vector objects based on attribute information. This type of script is useful for imported vector objects such as DLG and TIGER data that use standard tables, table relationships, and methods to attach attributes to elements that are duplicated in thousands of files (where each file represents a different geographic location) as a way to store and organize massive amounts of geodata. Many times it is impractical to view and/or use information directly from these large files because they map many different features with vast amounts of attribute information. Using all this data at once slows down the computer and takes focus away from the information you want to convey. For instance, a well-designed electronic atlas requires that you use only relevant geodata not only to get the best processing speed but also for design clarity. The knowledge of how to create this type of script makes it possible to extract only the data that is specifically relevant to your needs. Furthermore, you have the ability to split a complicated vector object into multiple independent objects that can be used separately. Finally, since you can use any number of vector objects as input, the script shows a fast way to obtain analogous elements and attribute information from multiple files.

In this example, 93 separate TIGER / Line vector objects for each of the 93 counties in the state of Nebraska (660 Mb), each with 20 database tables in the line elements database, were used as input for the tiger.sml script. In 10 minutes the script created 1 vector object (668 Kb) that has city boundary polygon elements for the whole state with three attribute tables: City_Names, Polygon_ID, and POLYSTATS.



Sample scripts have been prepared to illustrate how you might use the features of TNTmips' Spatial Manipulation Language (SML). If possible, the full script is printed below for your quick perusal. When a script is too long to fit on one page, key sections are reproduced below. The sample script illustrated can be downloaded from the SML script exchange at www.microimages.com/sml/ftpsmlink/TNT_Products_V6.5_CD.

Script for City Polygons (tiger.sml)

```
clear();
numInputs = GetInputVectorList(InputList,
    "Input Tigers");
if (numInputs <= 0) return;

GetOutputVector(Output, "VectorToolkit, Polygonal");
CopySubobjects(InputList[1], Output, "GEOREF");

Array xarray[1];
Array yarray[1];
numeric numpolys;
numpolys = 1;
Array records[1];
numeric numberofthem;
fieldname$ = "City_Names";

db = OpenVectorPolyDatabase(Output);
tinfo = TableCreate(db, "CITY_NAMES", "Created by
    SML script");
tinfo.OneRecordPerElement = 1;
TableAddFieldString(Output.poly.CITY_NAMES,
    fieldname$, 40);

for i = 1 to numInputs {
    SetStatusMessage(sprintf("Processing vector %d
        of %d\n", i, numInputs));

    linedb = OpenVectorLineDatabase(InputList[i]);
    if (TableExists(linedb, "Geo_Names_P") > 0) {
        linevar = TableOpen(linedb, "Geo_Names_P");
        linetable = TableGetInfo(linevar);

        numLines = NumVectorLines(InputList[i]);
        for j = 1 to numLines {
            SetStatusBar(j, numLines);

            left = (InputList[i].line[j].Basic_Data.
                FIPS_Pub55Pla_L);
            right = (InputList[i].line[j].Basic_Data.
                FIPS_Pub55Pla_R);

            if (left != right) {

                numPoints =
                    GetVectorLinePointList(InputList[i],
                        xarray, yarray, j);

                VectorAddLine(Output, numPoints,
                    xarray, yarray);
            }
        }
    }
}
```

Open Select Objects window and get input TIGER vector objects

Creates Output vector object with same Georeference as 1st input vector

Create database for polygon element with CITY_NAMES table / CITY_NAMES field

For every input vector...

Open Geo_Names_P table in input object

For every line in vector...

If line is a city boundary.. (In TIGER data, if the left and right side of a line have different attributes then the line is a city boundary.)

Get list of points in line

Add line to output vector

If the addition of a line creates a polygon (there are more polygons in output vector than there were in last iteration), then add attribute record to CITY_NAMES table for polygon element

```
if (NumVectorPolys(Output) >= numpolys) {
    numberofthem = TableReadAttachment(linetable,
        j, records);

    string$ = TableReadFieldStr(linetable,
        "Geographic_Name", records[1]);

    recordnumber = TableNewRecord(tinfo,
        string$);

    records[1] = recordnumber;

    numberofthem = TableWriteAttachment(tinfo,
        numpolys, records, 1);

    numpolys = numpolys + 1;
}
}
```

Get city name from Geographic_Name field

Add new record to table

Attach record to element

Polygon element ID number

```
Array islands[1];
numeric size;
size = 100;
Array deleteisland[size];
numeric numofislands;
numeric k;
k = 1;
```

Delete island polygons and attached records

```
for each poly[i] in Output {
    numofislands = GetVectorPolyIslandList(Output,
        islands);
    if (numofislands > 0) {
        for j = 1 to numofislands {
            TableReadAttachment(tinfo, islands[j],
                records);
            RecordDelete(tinfo, records[1]);
            deleteisland[k] = islands[j];
            k += 1;
            if (k >= size) {
                ResizeArrayPreserve(deleteisland, size+100)
                size += 100;
            }
        }
    }
}

if (k > 1)
    VectorDeletePolys(Output, deleteisland, k-1);

print("The number of island polygons deleted
    was", k-1);

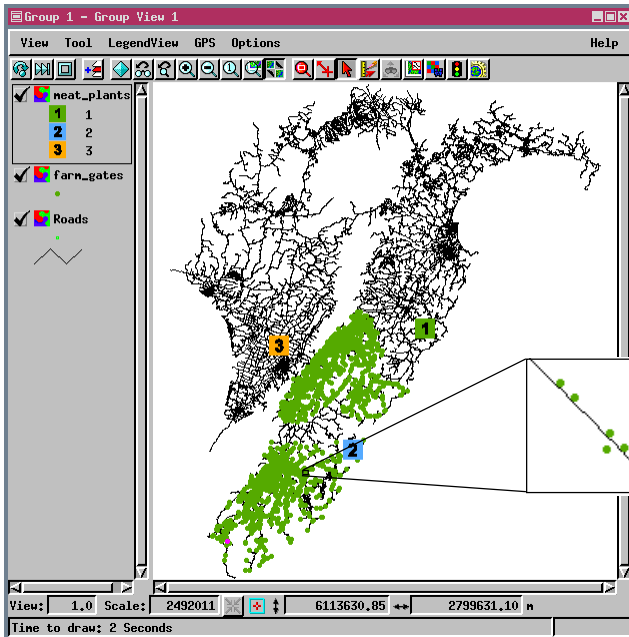
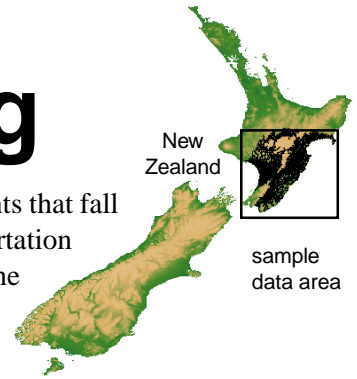
SetStatusBar(0, 10);

VectorValidate(Output);
```

Sample SML Script

Farm to Market Routing

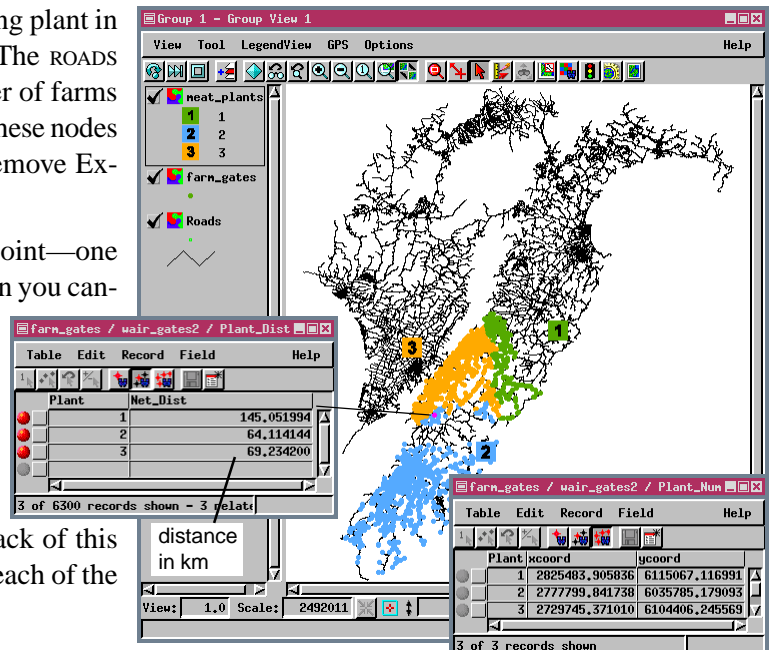
TNTmips includes a Network Analysis process that determines the “best” route between points that fall along a set of lines or the allocation of lines for the most efficient use in delivery to or transportation from a set of centers. In either case, the stops along the route or the centers must fall on the line network (the process automatically chooses the nearest node when you indicate the location of a stop or center). But what do you do when your points, in this case farm gates and processing plants, are not actually on the roads? Use an SML script like the one described here.



This script uses three vector objects: one to provide the road network, one with farm gate locations, and one with processing plant locations. You can substitute any widely distributed product location for the farm gates and any central location to which the product would be delivered for the processing plants. The difficulty in this particular case is that the data is not suitable for direct use in the Network Analysis process because, even if merged into a single vector object, the points do not fall on the roads (you may have to zoom in quite a way for it to be evident, see inset at left). The problem data was provided by AgriQuality New Zealand (formerly part of the Ministry of Agriculture and Forestry). This script uses only the distance from the processing plants to calculate impedance, but you can include a variety of other factors, such as road conditions, speed limits, and the price offered at each processing plant. You can readily change the market components of the impedance on a daily basis if need be with the end result of a dynamic appraisal of the best market for delivery of your product today.

The script adds a node to the ROADS object at the closest point on the closest line for each of the points in the FARMS object. It keeps track of these added nodes in an array that associates them with the correct farm. Nodes are similarly added for each of the processing plants. The shortest distance between each farm and processing plant is calculated using network analysis functions. This script adds two new tables to the point database of the FARMS vector object: one with records attached to each point that list the distance to each of the processing plants and another that provides the geographic coordinates of each processing plant in the same coordinate system used by the FARMS object. The ROADS vector ends up with many new nodes (equal to the number of farms and processing plants) that are not required for topology. These nodes can easily be removed by filtering the vector (use the Remove Excess Nodes filter) if desired.

The script attaches multiple records to each farm gate point—one for each processing plant. Such multiple attachments mean you cannot simply style by attribute because the first record attached to every point reports the distance to the first processing plant. In order to style each point according to which processing plant is closest or can be reached with the least impedance, you need to style by script using a script designed to evaluate all attached records. The script used to style the results shown here is included on the back of this page along with the script that determines the distance to each of the processing plants.



Sample scripts have been prepared to illustrate how you might use the features of TNTmips' Spatial Manipulation Language (SML). If possible, the full script is printed below for your quick perusal. When a script is too long to fit on one page, key sections are reproduced below. The sample script illustrated can be downloaded from the SML script exchange at www.microimages.com/sml/ftpsmlink/TNT_Products_V6.5_CD.

Script for Market Routes (network.sml)

```
clear(); #clear console
```

```
GetInputVector(Farms); #this vector is modified since tables are written to it
GetInputVector(Plants);
GetInputVector(TempNetwork); #this vector is modified by adding nodes
```

prompts for input vectors

```
VectorToolkitInit(TempNetwork,"NoDBStatTable");
VectorToolkitInit(Farms);
```

```
numeric numPoints;
numeric numFarms;
numeric numPlants;
numeric numLines;
numeric i;
```

variable declarations

```
Array xarray[1];
Array yarray[1];
```

```
numFarms = NumVectorPoints(Farms);
Array farms[numFarms];
numeric linenumber;
numeric tempx;
numeric tempy;
numeric a;
numeric b;
numeric distance;
```

gets georeference for farms and roads

```
farmgeo = GetLastUsedGeorefObject(Farms);
tempgeo = GetLastUsedGeorefObject(TempNetwork);
```

```
printf("The number of farms is %d\n",numFarms);
```

```
for i=1 to numFarms {
  SetStatusMessage(sprintf("Processing point %d of %d of farms",i,numFarms));
  tempx = Farms.point[i].Internal.x;
  tempy = Farms.point[i].Internal.y;
  GeorefTrans(farmgeo,tempx,tempy,tempgeo,tempx,tempy);
  linenumber = FindClosestLine(TempNetwork,tempx,tempy);
  ClosestPointOnLine(TempNetwork,linenumber,tempx,tempy,a,b);
  VectorAddNode(TempNetwork,a,b,1);
  farms[i] = FindClosestNode(TempNetwork,a,b);
}
```

finds closest point on closest line to farm gate and adds a node

creates array of all added nodes and corresponding farms

```
numPlants = NumVectorPoints(Plants);
Array plants[numPlants];
```

```
plantgeo = GetLastUsedGeorefObject(Plants);
```

```
printf("The number of plants is %d\n",numPlants);
```

```
for i=1 to numPlants {
  SetStatusMessage(sprintf("Processing point %d of %d of plants",i,numPlants));
  tempx = Plants.point[i].Internal.x;
  tempy = Plants.point[i].Internal.y;
  GeorefTrans(plantgeo,tempx,tempy,tempgeo,tempx,tempy);
  linenumber = FindClosestLine(TempNetwork,tempx,tempy);
  ClosestPointOnLine(TempNetwork,linenumber,tempx,tempy,a,b);
  VectorAddNode(TempNetwork,a,b,1);
  plants[i] = FindClosestNode(TempNetwork,a,b);
}
```

finds closest point on closest line to processing plant and adds a node

updates standard attributes and closes modified vector object (road network)

```
VectorUpdateStdAttributes(TempNetwork);
CloseVector(TempNetwork); #flush vector
```

```
class Network net;
class Route route;
class MultiRoute multiroute;
numeric imp;
```

```
net =
NetworkInit(GetObjectName(TempNetwork),GetObjectName(GetObjectName(TempNetwork),GetObjectName(TempNetwork)));
NetworkSetDefaultAttributes(net);
```

```
numLines = NumVectorLines(TempNetwork);
for i=1 to numLines {
  imp = (TempNetwork.line[i].LINESTATS.Length);
  NetworkLineSetImpedance(net,i,imp,"FromTo");
  NetworkLineSetImpedance(net,i,imp,"ToFrom");
}
```

sets network impedance to line length

```
total = numFarms * numPlants;
numeric count;
count = 1;
string tablename$;
class DATABASE db;
class DBTABLEINFO tinfo;
```

```
db = OpenVectorPointDatabase(Farms);
numeric recordnumber;
Array records[1];
numeric distance;
```

```
tinfo = TableCreate(db,"Plant_Num","Created by SML script");
TableAddFieldInteger(tinfo,"Plant",3);
TableAddFieldFloat(tinfo,"xcoord",25.6);
TableAddFieldFloat(tinfo,"ycoord",25.6);
for j=1 to numPlants {
  tempx = Plants.point[j].Internal.x;
  tempy = Plants.point[j].Internal.y;
  GeorefTrans(plantgeo,tempx,tempy,tempgeo,tempx,tempy);
  recordnumber = TableNewRecord(tinfo,j,tempx,tempy);
  records[1] = recordnumber;
  TableWriteAttachment(tinfo,i,records,1);
}
```

creates table that reports plant locations in same coordinate system as farms

```
tablename$ = "Plant_Dist";
tinfo = TableCreate(db,tablename$,"Created by SML Script");
TableAddFieldInteger(tinfo,"Plant",3);
class DBFIELDINFO the_field;
the_field = TableAddFieldFloat(tinfo,"Net_Dist",25.6);
the_field.UnitType = "Distance";
the_field.Units = "kilometers";
```

creates table and field with km distance units

```
for j=1 to numPlants {
```

```
  printf("Plant %d\n",j);
  SetStatusMessage(sprintf("Calculating all routes from plant %d of %d",j,numPlants));
  NetworkCalculateMultiRoute(net,plants[j],farms,numFarms,multiroute);
```

```
for i=1 to numFarms {
  SetStatusMessage(sprintf("Calculating route %d of %d",count,total));
  count +=1;
  printf("Route from plant %d to farm %d\n",j,i);
```

calculates distance of best route from each farm to each processing plant and adds to record attached to farm in distance table

```
  NetworkMultiRouteGetRoute(multiroute,farms[i],route);
  report$ = NetworkRouteGetReport(route);
```

```
  distance = StrToNum(GetToken(report$, " ",18));
  recordnumber = TableNewRecord(tinfo,j,distance/1000); #m/1000 = km
  records[1] = recordnumber;
  TableWriteAttachment(tinfo,i,records,1);
```

```
  NetworkRouteClose(route);
}
```

```
  NetworkMultiRouteClose(multiroute);
}
```

```
NetworkClose(net);
```

```
CloseVector(Farms);
CloseVector(TempNetwork);
CloseVector(Plants);
```

```
printf("Script Ran to Completion");
```

Script for Styling by Closest Plant

```
val = Plant_Dist[1].Net_Dist
id = Plant_Dist[1].Plant
```

```
for i = 2 to SetNum(Plant_Dist[*]) {
  if (Plant_Dist[i].Net_Dist < val) {
    val = Plant_Dist[i].Net_Dist;
    id = Plant_Dist[i].Plant;
  }
}
```

finds processing plant with lowest distance value

```
if (id == 1)
  Style$ = "Style1" else
if (id == 2)
  Style$ = "Style2" else
  Style$ = "Style3"
UseStyle = 1
```

assigns drawing style according to plant identified as closest

Sample SML Scripts

Movie Generation Scripts



The Spatial Manipulation Language (SML) now puts you in the director's chair. Using new movie generation SML functions in TNTmips 6.50 you can create scripts that set up and record custom animations of your geospatial data. You can record these animations in either MPEG format (any computer platform) or AVI format (Windows computers only) and set both frame rate and recording time. MicroImages has prepared a number of sample movie generation scripts, one of which is excerpted on the reverse side of this page. Although these movie generation scripts were prepared after the TNTmips 6.50 Products CD was mastered, you can download any of the scripts as well as sample movie files from the Downloads page of the MicroImages web site:

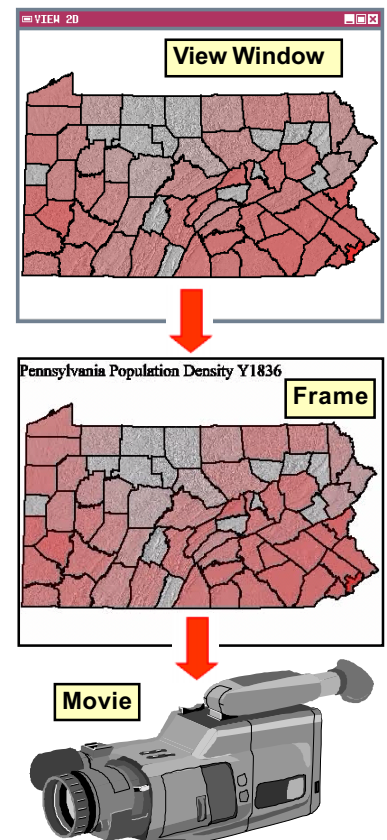
www.microimages.com/freestuf/

The 30 new SML functions and class methods that implement movie generation incorporate many of the capabilities of the 3D Simulation process in TNTmips. But they also give you more control over the 3D viewing parameters. You can specify viewer position and view direction independently, so that the 3D view can look ahead along the flight path (as in the standard 3D Simulation process), or to the side, straight down, or backward along the flight path. You can create a single 3D movie that incorporates

elements of the path, orbit, and pan modes of the 3D Simulation process.

But animations are not limited to 3D simulations. An animation merely consists of a gradually varying series of static frames. Each frame is rendered from one or more View windows created by the script and then copied into the output MPEG or AVI file. The movie therefore can record any sequential change in the view windows used to create the frames.

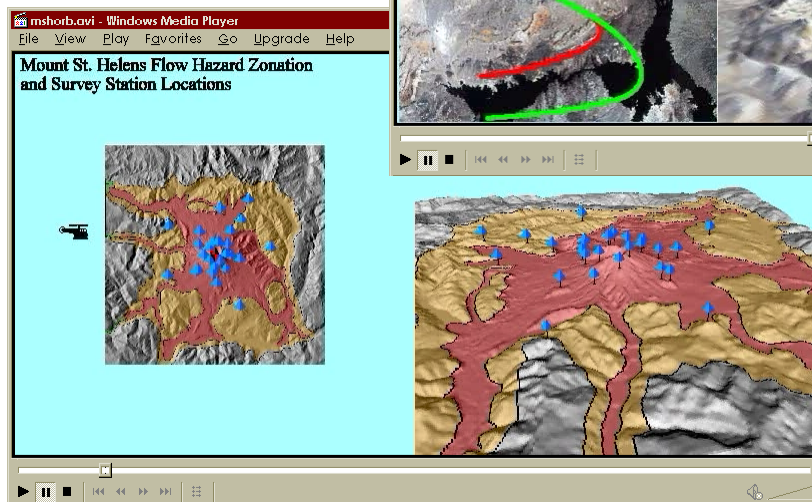
A simple example would be a sequence of 2D views showing a change in some mapped parameter through time. The script can sequentially add and then remove a series of pre-prepared layers to and from the view, or modify the display parameters for a single continuing layer (such as a set of vector polygons with attached attributes) to create the change from frame to frame. More complex examples might sequentially display the result of some process computed in the script, such as a series of viewsheds computed for different positions along a traverse line through a terrain model. The possibilities are limited only by your source data and your imagination!



The main processing loop of any movie script must perform three major functions:

1. Change the content of one or more view windows and redraw. This step might involve altering the viewing parameters for a 3D view, or adding new layers or updating existing layers in a 2D view.
2. Copy the contents of the view window(s) to a frame. Additional graphic symbols or annotation text can be drawn directly into the frame if you wish.
3. Copy the frame to the output movie file.

New functions in the Frame and Movie function groups are used to set up the generic frame and movie parameters, capture the View window contents to a frame, and copy the frame to the output AVI or MPEG file. For 3D animations, new class methods in the VIEWPOINT3D class are used to manipulate the settings for the 3D view. You can set viewer and view center position coordinates explicitly for each frame, or move either position a specified distance or direction relative to the previous position. Either position can be rotated around the other. You can also set either position and then use an azimuth angle, elevation angle, and distance to define the other.



In a 3D simulation script you can render both 2D and 3D views into each frame, as shown in the above illustration. After each frame is captured you can also draw symbols into it marking the current viewer position and view center position, and draw a trail of previous position symbols behind the moving symbol. Sample 3D simulation scripts are available to show how to script these features, and to set up panning, a spiral orbit, and constant-altitude and constant-height flight paths.

Sample scripts have been prepared to illustrate how you might use the new features of TNTmips' Spatial Manipulation Language (SML). Key sections of one script are reproduced below for your quick perusal. The entire script can be downloaded from the SML script exchange at www.microimages.com/sml/ftpsmlink/TNT_Products_V6.5_CD.

Excerpts from Constant Height Flight Path Script (PATHcHT2.sml)

```

string format$;
format$ = "AVI";

string framerate$;
framerate$ = "MOVIE_FRAMERATE_24";

numeric time;
time = 60;

GetInputRaster(Surface);
GetInputRaster(RastDrape);
GetInputVector(FlightPathVec);
GetInputVector(ViewCenterVec);

string styleFilename$;
string styleObjectName$;
GetInputObject("Style","Select style object for center and viewer
point symbols:", styleFilename$, styleObjectName$)

string viewer$;
viewer$ = "VIEWER";
string center$;
center$ = "CENTER"

print("START");

numeric size;
size = 320;

numeric zoomfactor;
zoomfactor = 1.0;

class GROUP group;
group = GroupCreate();

string flags$;
flags$ = "NoScalePosLine,NoIconBar,NoScrollbars,NoStatusLine";

class XmForm dialog2d;
class VIEW view2d;
dialog2d = CreateFormDialog("VIEW 2D");
view2d = GroupCreateView(group,dialog2d,"",size,size,flags$);
view2d.BackgroundColor.red = 67;
view2d.BackgroundColor.green = 100;
view2d.BackgroundColor.blue = 100;

class XmForm dialog3d;
class VIEW3D view3d;
dialog3d = CreateFormDialog("VIEW 3D");
view3d = GroupCreate3DView(group,dialog3d,"",size,size,flags$);
view3d.BackgroundColor.red = 67;
view3d.BackgroundColor.green = 100;
view3d.BackgroundColor.blue = 100;

GroupQuickAddRasterVar(group,Surface,1);
GroupQuickAddRasterVar(group,RastDrape,0);

DialogOpen(dialog2d);
DialogOpen(dialog3d);

ViewRedrawFull(view2d);
ViewRedrawFull(view3d);
ViewZoomOut(view2d,zoomfactor,1);

x2d = 0;
y2d = 0;
x3d = size;
y3d = 0;
w = 2 * size;
h = size;

numeric fontsize;
fontsize = 16;

class Frame frame;
frame = FrameCreate(w,h);

ActivateGC(FrameCreateGC(frame));
DrawTextSetHeightPixels(fontsize);
DrawUseStyleObject(styleFilename$,styleObjectName$);

class Movie movie;
movie = MovieInit();

MovieSetFormat(movie,format$);
MovieSetFrameRate(movie,framerate$);
MovieSetFrameWidth(movie,w);
MovieSetFrameHeight(movie,h);

string ext$;
ext$ = MovieGetFileExt(movie);
string filename$;
filename$ = GetOutputFileName("","Make filename for movie",ext$);

if (time <= 1.0) time = 1.0;

numFrames = time * rate;

class Georef georefS;
georefS = GetLastUsedGeorefObject(Surface);
GeorefSetProjection(georefS,group.Projection);

class Georef georefFlight;
georefFlight = GetLastUsedGeorefObject(FlightPathVec);
GeorefSetProjection(georefFlight,group.Projection);

class Georef georefCent;
georefCent = GetLastUsedGeorefObject(ViewCenterVec);
GeorefSetProjection(georefCent,group.Projection);

MovieStart(movie,filename$);

for i = 1 to numFrames {
class POINT3D fpt;
fpt.x = xarrayf_eq[i];
fpt.y = yarrayf_eq[i];
fpt.z = zarrayf_eq[i];
vp.SetViewerPosition(fpt);

class POINT3D cpt;
cpt.x = xarrayc_eq[i];
cpt.y = yarrayc_eq[i];
cpt.z = zarrayc_eq[i];
vp.SetCenter(cpt);

ViewRedraw(view3d);

FrameCopyFromView(frame,view2d,0,0,size,size,x2d,y2d);
FrameCopyFromView(frame,view3d,0,0,size,size,x3d,y3d);

for j = 1 to (i - 1) {
SetColor(colorc)
FillCircle(xarraycs[j],yarraycs[j],2)
}

class POINT2D point;
point.x = vp.CenterPoint.x;
point.y = vp.CenterPoint.y;
point =
TransPoint2D(point,ViewGetTransMapToView(view2d,group.Projection));
point = TransPoint2D(point,ViewGetTransViewToScreen(view2d));
DrawSetPointSize(center$);
DrawPoint(point.x,point.y);

xarraycs[i] = point.x;
yarraycs[i] = point.y;

for j = 1 to (i - 1) {
SetColor(colorf)
FillCircle(xarrayfs[j],yarrayfs[j],2)
}

point.x = vp.ViewPos.x;
point.y = vp.ViewPos.y;
point =
TransPoint2D(point,ViewGetTransMapToView(view2d,group.Projection));
point = TransPoint2D(point,ViewGetTransViewToScreen(view2d));
DrawSetPointSize(viewer$);
DrawPoint(point.x,point.y);

xarrayfs[i] = point.x;
yarrayfs[i] = point.y;

DrawTextSetColors(black);
DrawTextSimple("Muddy Mountains, NV",2,fontsize);
DrawTextSetColors(colorc);
DrawTextSimple("View center path",2,fontsize*2.1);
DrawTextSetColors(colorf);
DrawTextSimple("Flight path",2,fontsize*3.3);

MovieAddFrame(movie,frame);
}

MovieStop(movie);
MovieExit(movie);
DialogClose(dialog2d);
DialogClose(dialog3d);

```

Set movie format, frame rate, and recording time

Select input DEM for surface, raster drape, and two vector objects containing ground traces of flight path and view center path

Select style object containing point symbol styles for viewer position and view center position

Variables to set size of 2D and 3D view windows and zoom-out factor for 2D view

Create display group. Create flag to create view without iconbar, scrollbars, status line, and scale/position line; important to maintain fixed window size during movie generation

Create dialog and 2D view

Create dialog and 3D view

Add surface and raster drape to group

Open both views

Full redraw of both views

Parameters to set location and size of each view in movie frame

Set fontsize for text annotation in frame

Create blank frame

Create graphics context for frame

Initialize movie

Set more movie parameters

Make output file

Check recording time and calculate number of frames

Get georeference parameters for layers and reset to group projection defined by raster drape layer

Start recording movie

Section computing arrays of viewer and center positions from input vectors is omitted here; see script

Begin loop for each frame

Set viewer position along flight path

Set view center position

Redraw both views

Copy both views to frame

Loop to draw previous center points in frame

Draw current center point in frame

Update arrays of previous center point coordinates

Loop to draw previous viewer positions in frame

Draw current viewer position in frame

Update arrays of previous viewer position coordinates

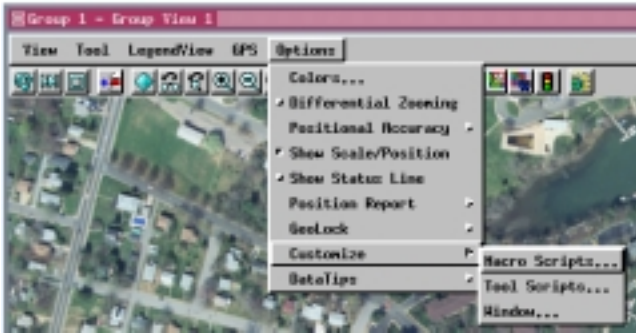
Draw text annotation in frame

Add frame to movie

End of loop recording frames

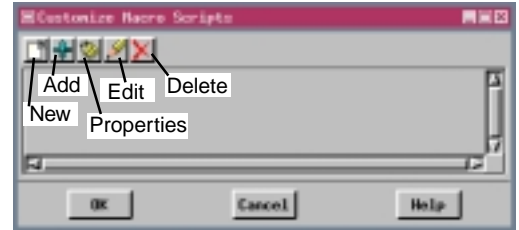
Stop and exit movie, close dialogs

Macro Script Setup

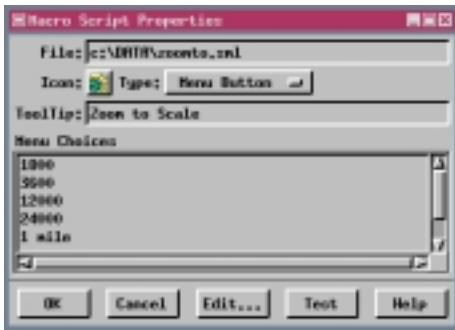


Macro scripts and tool scripts add a powerful new way to use Spatial Manipulation Language (SML) in your TNT products. To add a macro script so it can be run from an icon on the View window toolbar, choose Options / Customize / Macro Scripts from the View window in the Display process or any other process with a View window. Making this selection opens the Customize Macro Scripts window.

Click on New if your script is not yet written or click on Add if you already have a script

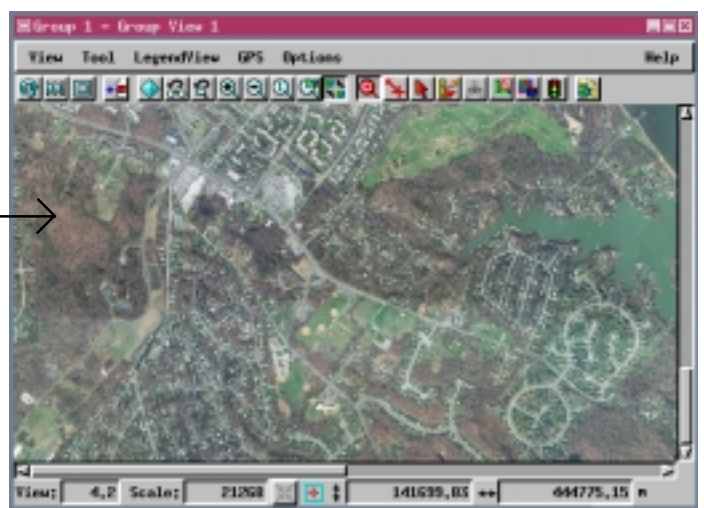
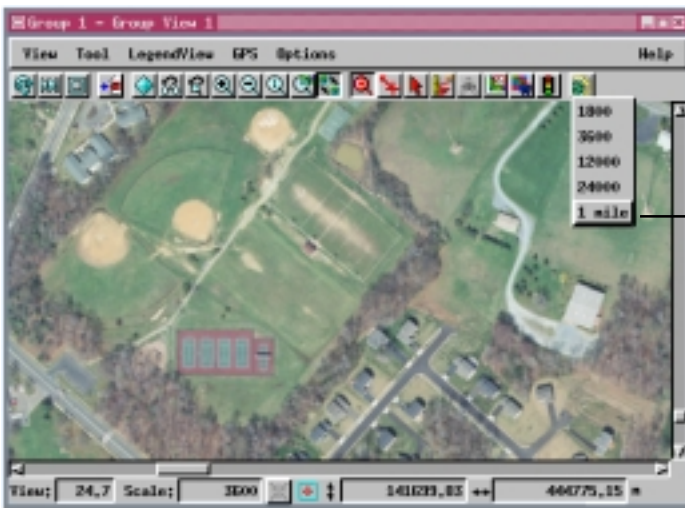


(Process / SML / Edit Script). Clicking on New opens the Query Editor window, which is used to prepare / edit SML scripts. A few predefined symbols that may be of use appear as comments when the Query Editor opens. The same features available for script construction in the SML process are available here. You are prompted to save your script when you click on OK in the Query Editor. Once the script is saved, the Macro Script Properties window opens. Clicking on Add opens the Select File window so you can locate the script you want to use. Only *.sml files are listed by default—you need to change the *Files of Type* option to all if your script has a different file extension or has been saved as an RVC object. Once the script is located and you click OK, the Macro Script Properties window opens.



The Macro Script Properties window lets you choose an icon, indicate whether the script is for a simple button or a menu button, set up the ToolTip, enter menu items if a menu button is used, and test your script. Clicking on the Icon button opens the Select Icon window so you can choose a different icon from the more than 700 icons in this size available with TNTmips. The Type option menu offers two choices: Simple Button and Menu button. A simple button automatically executes its script without further input from the user. A menu button drops down a list of choices that determine the outcome of running the script. If Menu Button is chosen in the Macro Script Properties window, the Menu Choices text field becomes active so you can enter the menu choices you want available when you click on the Macro Script icon on the View toolbar. Each line in the Menu Choices field represents a separate menu choice. Enter the ToolTip you want directly in the ToolTip field. This ToolTip will appear when the cursor is paused over the Macro Script icon in the View window. The Test button at the bottom of the window lets you run your script without closing the customize windows. If the script uses a menu button, the menu choice with the text cursor is the option chosen. If the script does not perform exactly as anticipated, the Edit button at the bottom of the window opens the Query Editor containing the script so you can make modifications.

Click OK in the Macro Script Properties and Customize Macro Scripts windows when you are done adding, developing, and/or testing your script. Return to the Customize Macro Scripts window anytime to add or delete scripts, or open the Macro Script Properties window (click on Properties icon in Customize Macro Scripts window) to change the icon, ToolTip, or menu choices.



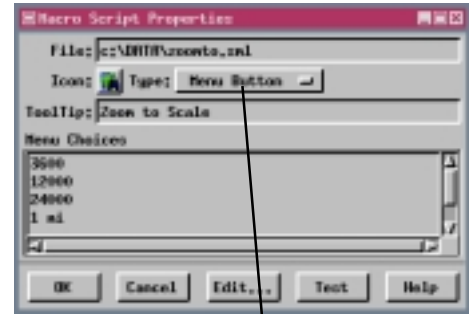
Sample SML Macro Script

Zoom to Specified Map Scale

The Zoom to Scale macro script lets you specify a number of different map scales or ground dimensions to use to adjust the display scale. When you pick the desired scale from the drop-down menu on the Zoom to Scale script icon, the contents of the View window are redrawn at the indicated scale.



If the scale is specified in ground dimensions, for example 1 mile or 10 miles, you get the specified distance plus 10% for the smallest window dimension. (For example, if you choose 10 miles and the window is wider than it is tall, 11 miles is visible in the vertical dimension of the window.)



This option menu lets you designate whether your macro script is designed for a simple button (push the button and the macro script is executed) or a menu button. In order for a macro script to provide choices, you must indicate you want the icon to be a menu button. You can then type in the menu choices you want.

In order for the Zoom to Scale script to work as designed, the objects displayed must either be georeferenced or scale calibrated. In order for map scale displays to be accurate, you also need to have set up your screen width and height on the MI/X panel for General System Preferences (Support / Setup / Preferences).

This particular script accepts two kinds of scale input, map scale and miles. If the menu choice is purely numeric, it is taken as a designated map scale. If the menu choice consists of a number, a space, and any other character, it is taken as a designated ground distance in miles.



Each time you select from the Zoom to Scale icon menu, the scale is adjusted while maintaining the same view center if possible (if you are zooming out and are near the edge of the displayed objects, the view will be recentered).

Remember when working with map scale that a smaller entered number means things appear larger. For example, the dimensions of a park or other area of interest will be twice as large at 1:12000 as they are at 1:24000, which are represented by menu choices of 12000 and 24000, respectively.



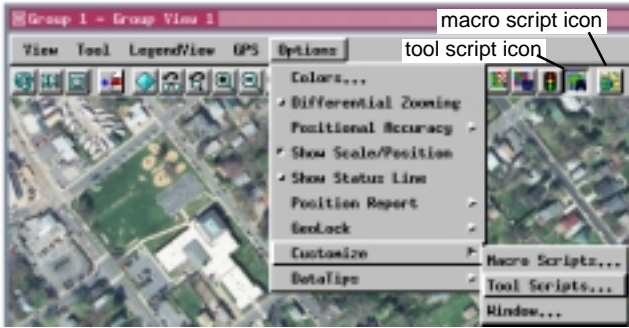
Macro and Tool Scripts can be created using SML in any TNTmips process that uses a View window (Options / Customize from the View window menu bar). These scripts are then available from an icon, which you select or design, on the toolbar. Sample scripts have been prepared to illustrate how you might use these features, which are available only in TNTmips 6.4 or later, to assist with specific tasks you perform on a regular basis. If possible, the full script is printed below for your quick perusal. When a script is too long to fit on one page, key sections are reproduced below. All sample Tool and Macro Scripts illustrated can be found in their entirety on your TNT products CD-ROM in the directory where your TNT products are installed. These scripts, among others, can be downloaded from the SML script exchange at www.microimages.com/sml/ftpsmllink/TNT_Products_V6.4_CD.

Script for Zoom to Scale (zoomto.sml)

```
if (NumberTokens(MenuChoice$, " ") == 1) {
    ViewSetMapScale(View, StrToNum(MenuChoice$));
}
else if (NumberTokens(MenuChoice$, " ") == 2) {
    widthmeters = View.PixelSizeMillimeters * View.Width / 1000;
    heightmeters = View.PixelSizeMillimeters * View.Height / 1000;
    if (widthmeters < heightmeters) {
        mindim = widthmeters;
    }
    else {
        mindim = heightmeters;
    }
    newdim = StrToNum(GetToken(MenuChoice$, " ", 1)) * GetUnitConvDist("miles", "meters");
    newscale = newdim / mindim * 1.1;
    ViewSetMapScale(View, newscale);
}
```

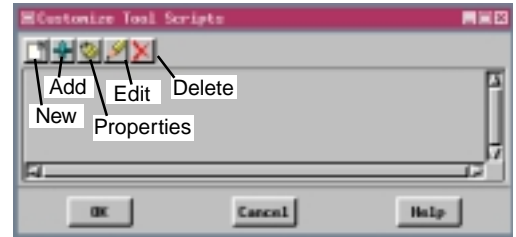
You could change this script to zoom to entered kilometer dimensions, rather than miles, by changing the word *miles* to *kilometers* here.

Tool Script Templates

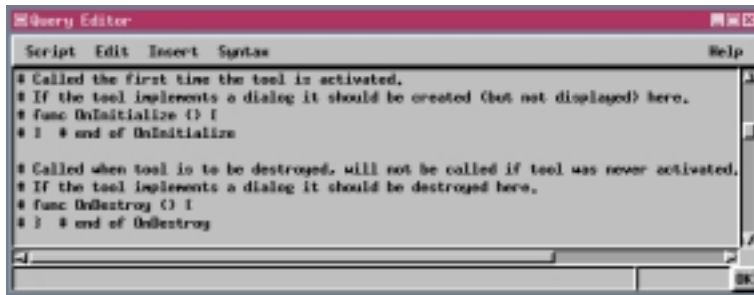


Tool scripts and macro scripts add a powerful new way to use Spatial Manipulation Language (SML) in your TNT products. To add a tool script to run from an icon on the View window toolbar, choose Options / Customize / Tool Scripts from the View window in the Display process or any other process with a View window. Making this selection opens the Customize Tool Scripts window. Tool Script icons appear to the left of any Macro Script icons on the View window toolbar.

Click on New if your script is not yet written or click on Add if you

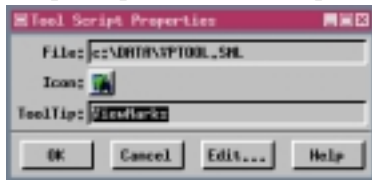


or someone else has already written the script (Process / SML / Edit Script). Clicking on New opens the Query Editor window, which is used to prepare / edit SML scripts. The same predefined symbols provided for macro scripts appear as comments when the Query Editor opens. Additionally, a number of predefined values (such as number PointerX, which provides the pointer X coordinate within the view in pixels) and functions likely to be used in a Tool Script are provided as a template for your custom script.



The template includes functions used the first time a tool is activated; when the tool is destroyed; when the tool is activated and deactivated; when the tool is suspended (during redraw) and resumed (after redraw); when the left, right, or middle mouse button is pressed or released; when the cursor moves without a button press; when the cursor moves with a button press; when the cursor enters or leaves the View window; and when the user presses a key. If you want to use these functions in your script, uncomment the lines (remove the leftmost #) and add function code between the lines as needed. The same features available for script construction in the SML process are available here. You are prompted to save your script when you click on OK in the Query Editor. Once the script is saved, the Tool Script Properties window opens.

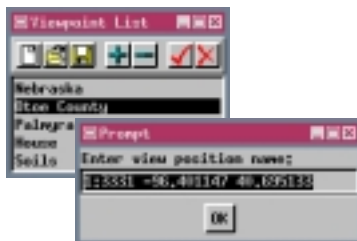
The Customize Tool Scripts window has exactly the same buttons and functions as the Customize Macro Scripts window (see the *Macro Script Setup* color plate). If you choose to add an existing script, once the script is located and you click OK, the Tool Script Properties window opens. The Tool Script Properties window lacks some of the features of the Macro Script Properties window—you choose an icon and set up the ToolTip, but there is no Type option button and consequently no Menu Choices panel. The Test button is not available for tool scripts. You need to test the tool from the View window itself once the tool is added.



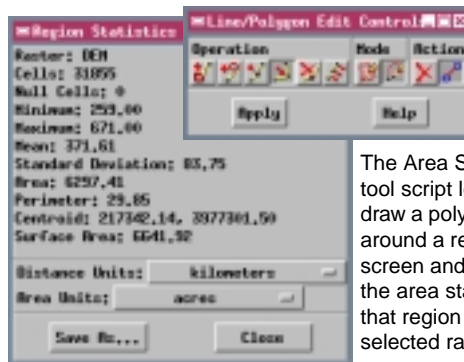
Click OK in both the Tool Script Properties and Customize Tool Scripts windows when you are done adding your script. You can always return to the Customize Tool Scripts window (Options / Customize / Tool Scripts) to add or delete scripts, or open the Tool Script Properties window (click on the Properties icon in Customize Tool Scripts window) to change the icon or ToolTip.

A number of different sample tool scripts are provided with the TNT products. You can use components from any or all of these scripts to create the custom tool you need for your specialized application. The interface windows created by some of these tool scripts are shown below.

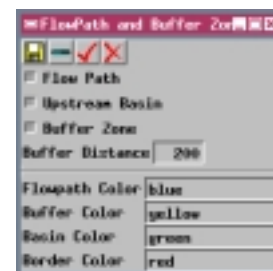
The ViewMarks tool script builds up a list of desired view points and scales so you can jump to specific locations.



The ViewMarks tool script builds up a list of desired view points and scales so you can jump to specific locations.



The Area Statistics tool script lets you draw a polygon around a region on the screen and computes the area statistics for that region of the selected raster.



The Flow Path tool script lets you choose which of a number of watershed properties you would like to see from a given point in the view.

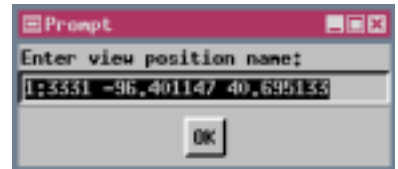
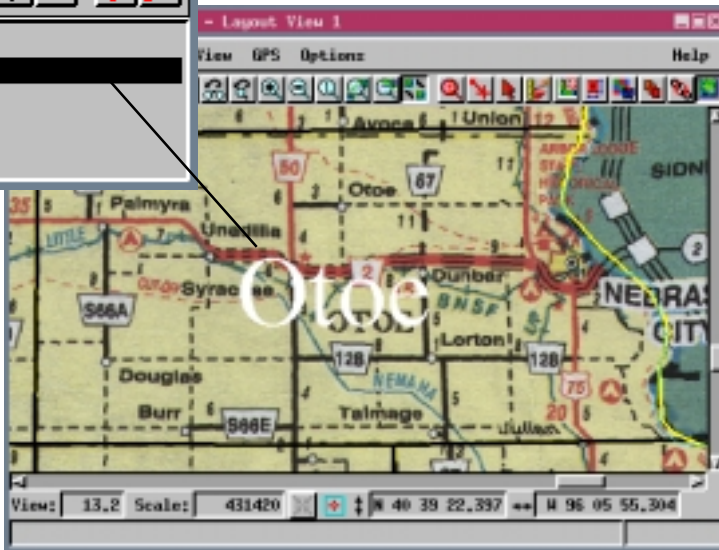
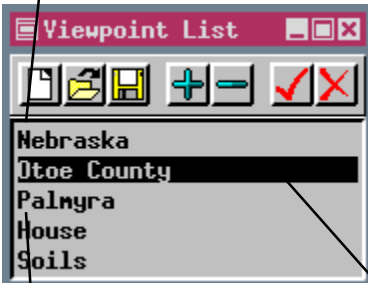
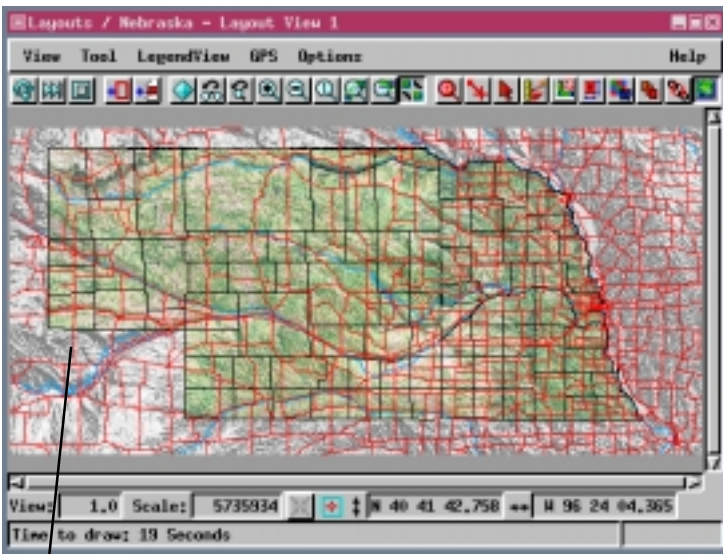
Sample SML Tool Script ViewMarks

ViewMarks are position markers for a single view window. They are particularly useful for layouts covering a large geographic area, especially when limited map scale visibility is used to add and remove layers as you zoom in and out. Mark a view of interest and return to that view from any scale or position by selecting it from the list of viewpoints you build up.

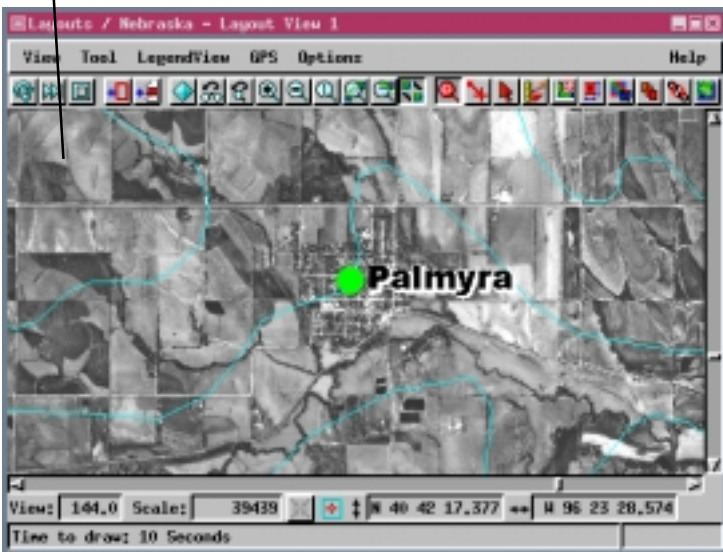
The script, a portion of which is shown on the other side of this page, creates the Viewpoint List window (below, left) with the buttons needed to make, save, and open

viewpoint lists; to add and remove points from the list; and to zoom to the selected point and close the window. You can also double-click on a list entry to display it.

When you add a ViewMark, the default name provided is the map scale and center coordinates for the view, which can be changed to a more descriptive name.



The Viewpoint List remains as long as the current View window is open. If you want to use the same ViewMarks in another display session, you need to save the list. When you choose to save your viewpoint list, a .pos file is created. This file simply contains the name you entered for the viewpoint, the map scale, and center point for each viewpoint on the list. Thus, the file can be used again with the same group, layout, or single-layout atlas, or it can be used with a completely different set of layers that covers the same geographic area.



Thus, ViewMarks let you work with data sets that cover large areas and still rapidly locate and return to areas of interest in high resolution imagery or detailed vector objects. ViewMarks have use beyond TNTmips' Display process; they can be set up for any process that uses a View window, for example, the Spatial Data Editor. Thus, you can mark a number of positions that are critical to check after some global editing operation, such as line snapping or filtering, and return to each with ease.

The example on this page is derived from the Nebraska Statewide atlas, which is a single layout that uses map scale controlled visibility to increase the level of detail shown as you zoom in.

Macro and Tool Scripts can be created using SML in any TNTmips process that uses a View window (Options / Customize from the View window menu bar). These scripts are then available from an icon, which you select or design, on the toolbar. Sample scripts have been prepared to illustrate how you might use these features, which are available only in TNTmips 6.4 or later, to assist with specific tasks you perform on a regular basis. If possible, the full script is printed below for your quick perusal. When a script is too long to fit on one page, key sections are reproduced below. All sample Tool and Macro Scripts illustrated can be found in their entirety on your TNT products CD-ROM in the directory where your TNT products are installed. These scripts, among others, can be downloaded from the SML script exchange at www.microimages.com/sml/ftpsmlink/TNT_Products_V6.4_CD.

Partial Script for ViewMarks (vptool.sml)

```

class XmForm dlgform;
class XmList poslist;
class MAPPROJ projLatLon;
class TRANSPARM transMapToView;
class FILE posfile;
number ischanged;
number setDefaultWhenClose;
number numpos;
array posX[1];
array posY[1];
array posScale[1];

func DoSave () {
    if (numpos == 0) return;
    posfilename$ = GetOutputFileName("", "Select position file to save as:", "pos");
    DeleteFile(posfilename$);
    posfile = fopen(posfilename$, "w");
    if (posfile == 0) return (false);
    local i;
    for i = 1 to numpos {
        fprintf(posfile, "%s, %f, %f, %f\n", poslist.GetItemAtPos(i), posX[i], posY[i], posScale[i]);
    }
    fclose(posfile);
    ischanged = false;
    return (true);
}

func AskSave () {
    if (!ischanged || numpos == 0) return (true);
    local answer;
    answer = PopupYesNoCancel("Save current point list?", 1);
    if (answer < 0) return (false);
    if (answer == 0) return (true);
    return (DoSave());
}

proc DoZoom () {
    local selpos;
    if (numpos == 0) return;
    selpos = poslist.GetFirstSelectedPos();
    if (selpos > 0) {
        transMapToView = ViewGetTransMapToView(View, projLatLon);
        if (transMapToView == 0) {
            PopupMessage("Cannot obtain map/view transformation.");
            return;
        }
        class POINT2D zpoint;
        zpoint.x = posX[selpos];
        zpoint.y = posY[selpos];
        zpoint = TransPoint2D(zpoint, transMapToView, false);
        class RECT vextents;
        vextents = View.Extents;
        if (zpoint.x < vextents.x1 || zpoint.x > vextents.x2 || zpoint.y < vextents.y1 || zpoint.y > vextents.y2) {
            PopupMessage("Point is outside extents of objects being viewed.");
            return;
        }
        View.DisableRedraw = true;
        View.CurrentMapScale = posScale[selpos];
        View.Center = zpoint;
        View.DisableRedraw = false;
        View.Redraw(View);
    }
}

proc DoAdd () {
    transMapToView = ViewGetTransMapToView(View, projLatLon);
    if (transMapToView == 0) {
        PopupMessage("Cannot obtain map/view transformation.");
        return;
    }
    class POINT2D cpoint;
    cpoint = TransPoint2D(View.Center, transMapToView, true);
    numpos = numpos + 1;
    ResizeArrayPreserve(posX, numpos);
    ResizeArrayPreserve(posY, numpos);
    ResizeArrayPreserve(posScale, numpos);
    posX[numpos] = cpoint.x;
    posY[numpos] = cpoint.y;
    posScale[numpos] = View.CurrentMapScale;
    namestr$ = sprintf("I:%.0f %f %f", posScale[numpos], posX[numpos], posY[numpos]);
    namestr$ = PopupString("Enter view position name:", namestr$);
    while (poslist.ItemExists(namestr$)) {
        namestr$ = PopupString("Name already used.\nEnter view position name:", namestr$);
    }
}

poslist.AddItem(namestr$);
ischanged = true;
}

proc DoRemove () {
    local selpos;
    local i;
    if (numpos == 0) return;
    selpos = poslist.GetFirstSelectedPos();
    if (selpos > 0) {
        poslist.DeletePos(selpos);
        for i = selpos to numpos - 1 {
            posX[i] = posX[i+1];
            posY[i] = posY[i+1];
            posScale[i] = posScale[i+1];
        }
        numpos = numpos - 1;
        ischanged = true;
    }
}

proc DoNew () {
    if (!AskSave()) return;
    numpos = 0;
    poslist.DeleteAllItems();
    ischanged = false;
}

proc DoOpen () {
    if (!AskSave()) return;
    posfile = GetInputTextFile("", "Select positions file to open:", "pos");
    if (posfile == 0) return;
    numpos = 0;
    poslist.DeleteAllItems();
    ischanged = false;
    while (!feof(posfile)) {
        filestr$ = fgetline$(posfile);
        if (NumberTokens(filestr$, ",") < 4) continue;
        numpos = numpos + 1;
        ResizeArrayPreserve(posX, numpos);
        ResizeArrayPreserve(posY, numpos);
        ResizeArrayPreserve(posScale, numpos);
        poslist.AddItem(GetToken(filestr$, ",", 1));
        posX[numpos] = StrToNum(GetToken(filestr$, ",", 2));
        posY[numpos] = StrToNum(GetToken(filestr$, ",", 3));
        posScale[numpos] = StrToNum(GetToken(filestr$, ",", 4));
    }
    fclose(posfile);
}

proc DoClose () {
    if (setDefaultWhenClose) {
        setDefaultWhenClose = false;
        View.SetDefaultTool();
    }
}

func OnInitialize () {
    class MAPPROJ tempLatLon;
    tempLatLon.System = "LatLon";
    tempLatLon.Datum = "WGS84";
    projLatLon = tempLatLon;
    dlgform = CreateFormDialog("Viewpoint List", View, Form);
    WidgetAddCallback(dlgform.Shell.PopdownCallback, DoClose);
    class PushButtonItem btnItemNew;
    class PushButtonItem btnItemOpen;
    class PushButtonItem btnItemSave;
    class PushButtonItem btnItemAdd;
    class PushButtonItem btnItemRemove;
    class PushButtonItem btnItemZoom;
    class PushButtonItem btnItemClose;
    btnItemNew = CreatePushButtonItem("New", DoNew);
    btnItemNew.IconName = "new";
    btnItemOpen = CreatePushButtonItem("Open...", DoOpen);
    btnItemOpen.IconName = "open_";
    btnItemSave = CreatePushButtonItem("Save...", DoSave);
    btnItemSave.IconName = "save";
    btnItemAdd = CreatePushButtonItem("Add", DoAdd);
    btnItemAdd.IconName = "add_sel";
    btnItemRemove = CreatePushButtonItem("Remove", DoRemove);
    btnItemRemove.IconName = "remove_sel";
    btnItemZoom = CreatePushButtonItem("Zoom", DoZoom);
    btnItemZoom.IconName = "apply";
    btnItemClose = CreatePushButtonItem("Close", DoClose);
    btnItemClose.IconName = "delete";
}

```

saves the list to a file

removes selected item from list

clears the list

opens file containing list

zooms to selected position

closes the window and switches to default tool

is called the first time the tool is activated

adds current viewpoint to list

(see vptool.sml for full script)

Macro and Tool Scripts can be created using SML in any TNTmips process that uses a View window (Options / Customize from the View window menu bar). These scripts are then available from an icon, which you select or design, on the toolbar. Sample scripts have been prepared to illustrate how you might use these features, which are available only in TNTmips 6.4 or later, to assist with specific tasks you perform on a regular basis. If possible, the full script is printed below for your quick perusal. When a script is too long to fit on one page, key sections are reproduced below. All sample Tool and Macro Scripts illustrated can be found in their entirety on your TNT products CD-ROM in the folder in which you installed TNTmips 6.4. These scripts, among others, can be downloaded from the SML script exchange at www.microimages.com/sml/ftpsmlink/TNT_Products_V6.4_CD.

Script for Area Statistics (regstats.sml)

```

proc cbRedraw() {
  local numeric larea, lperimeter, lsurface;

  larea = areaScale * area;
  lperimeter = distScale * perimeter;
  lsurface = areaScale * surface;
  if (gc == 0) return;
  ActivateGC(gc);

  SetColorName("gray75");
  FillRect(0, 0, da.width, da.height);
  SetColorName("black");
  if (cells > 0) {DrawInterfaceText(sprintf("Raster: %s\nCells:
%d\nNull Cells: %d\nMinimum: %.2f\nMaximum: %.2f\nMean:
%.2f\nStandard Deviation: %.2f\nArea: %.2f\nPerimeter:
%.2f\nCentroid: %.2f, %.2f\nSurface Area: %.2f",
rasterName$, count, cells - count, min, max, mean, stdDev, larea,
lperimeter, centroid.x, centroid.y, lsurface), 0, 10);
}
  else DrawInterfaceText(sprintf("Raster: %s\nCells:\nNullCells:
\nMinimum:\nMaximum:\nMean:\nStandardDeviation:\nArea:\nPerimeter:
\nCentroid:\nSurface Area:", rasterName$), 0, 10);
}
}

proc cbToolApply(class RegionTool tool) {
  if (checkLayer()) {
    local numeric sum, sumsq, xscale, yscale, zscale;
    local numeric current, right, down, downright;
    local region MyRgn;
    local class StatusHandle status;
    local class StatusContext context;
    cells = 0; min = 0; max = 0; mean = 0; stdDev = 0; sum = 0;
    sumsq = 0; count = 0; surface = 0; area = 0; perimeter = 0;
    centroid.x = 0; centroid.y = 0; current = 0; right = 0; down = 0;
    downright = 0;
    xscale = ColScale(targetRaster);
    yscale = LinScale(targetRaster);
    zscale = Group.ActiveLayer.zscale;

    MyRgn = tool.Region;
    MyRgn = RegionTrans(MyRgn, ViewGetTransLayerToScreen(View,
rasterLayer, 1));
    MyRgn = RegionTrans(MyRgn, ViewGetTransLayerToView(View,
rasterLayer));
    MyRgn = RegionTrans(MyRgn, ViewGetTransMapToView(View,
rasterLayer.Projection, 1));

    context = StatusContextCreate(status);
    StatusSetMessage(context, "Computing values...");

    foreach targetRaster[lin, col] in MyRgn {
      if (!IsNull(targetRaster)) {
        if (count == 0) {
          max = targetRaster;
          min = targetRaster;
        }
        else if (targetRaster > max) {
          max = targetRaster;
        }
        else if (targetRaster < min) {
          min = targetRaster;
        }
        sum += targetRaster;
        sumsq += sqr(targetRaster);
        count += 1;
      }

      if (!IsNull(targetRaster))
        current = targetRaster;
      if (!IsNull(targetRaster[lin,col+1]))
        right = targetRaster[lin,col+1];
      if (!IsNull(targetRaster[lin+1,col]))
        down = targetRaster[lin+1,col];
      if (!IsNull(targetRaster[lin+1,col+1]))
        downright = targetRaster[lin+1,col+1];
      surface += .5*sqrt(sqr(yscale*current*zscale-yscale*
right*zscale)+sqr(xscale*current*zscale-xscale*down*zscale)
+sqr(xscale*yscale));
    }
  }
}

```

scales the values to specified units

redraws Region Statistics window when new statistics are computed

main tool procedure, called within OnInitialize function

defines local variables for statistics calculation

creates status dialog

computes statistics for the polygon

estimates surface area by adding areas of triangles created by current cell and cell below, cell to right, and cell to lower right.

```

  surface += .5*sqrt(sqr(yscale*downright*zscale-yscale*right*zscale)+sqr(xscale*downright*zscale-
xscale*down*zscale)+sqr(xscale*yscale));
  cells += 1;
}
CloseRaster(targetRaster);

if (count > 1) {
  mean = sum / count;
  stdDev = sqrt((sumsq - sqr(sum) / count) / (count - 1));
  area = MyRgn.$Data.GetArea();
  perimeter = MyRgn.$Data.GetPerimeter();
  centroid = MyRgn.$Data.GetCentroid();
}
cbRedraw();

StatusContextDestroy(context);
StatusDialogDestroy(status);
} # end of cbToolApply
}

func OnInitialize () {
  form = CreateFormDialog("Region Statistics");
  form.marginHeight = 2;
  form.marginWidth = 2;
  WidgetAddCallback(form.Shell.PopdownCallback, cbClose);

  da = CreateDrawingArea(form, 173, 301);
  da.topWidget = form;
  da.leftWidget = form;
  da.rightWidget = form;
  WidgetAddCallback(da.ExposeCallback, cbRedraw);

  line1 = CreateHorizontalSeparator(form);
  line1.topWidget = da;
  line1.leftWidget = form;
  line1.rightWidget = form;
  line1.bottomOffset = 2;

  distMenu = CreateUnitOptionsMenu(form, "distance_units_c", cbDistUnits,
2, 0);
  distMenu.topWidget = line1;
  distMenu.leftWidget = form;

  areaMenu = CreateUnitOptionsMenu(form, "area_units_c", cbAreaUnits,
1, 0);
  areaMenu.topWidget = distMenu;
  areaMenu.leftWidget = form;

  line2 = CreateHorizontalSeparator(form);
  line2.topWidget = areaMenu;
  line2.leftWidget = form;
  line2.rightWidget = form;
  line2.topOffset = 2;

  saveButton = CreatePushButtonItem("Save As...", cbSave);
  closeButton = CreatePushButtonItem("Close", cbClose);

  buttonRow = CreateButtonRow(form, saveButton, closeButton);
  buttonRow.topWidget = line2;
  buttonRow.leftWidget = form;
  buttonRow.rightWidget = form;
  buttonRow.bottomWidget = form;

  tool = ViewCreatePolygonTool(View, "", "", "");
  ToolAddCallback(tool.ActivateCallback, cbToolApply);
} # end of OnInitialize

func OnDestroy () {
  tool.Managed = 0;
  DestroyGC(gc);
  DestroyWidget(form);
} # end of OnDestroy

```

avoids division by zero when computing mean and standard deviation.

destroys the status dialog when computations are complete

is called the first time the tool is activated; creates the graphic tool and dialog

creates drawing area for dialog window

creates separator between statistics and menus

creates menu for selecting units

creates separator between menus and buttons

creates buttons

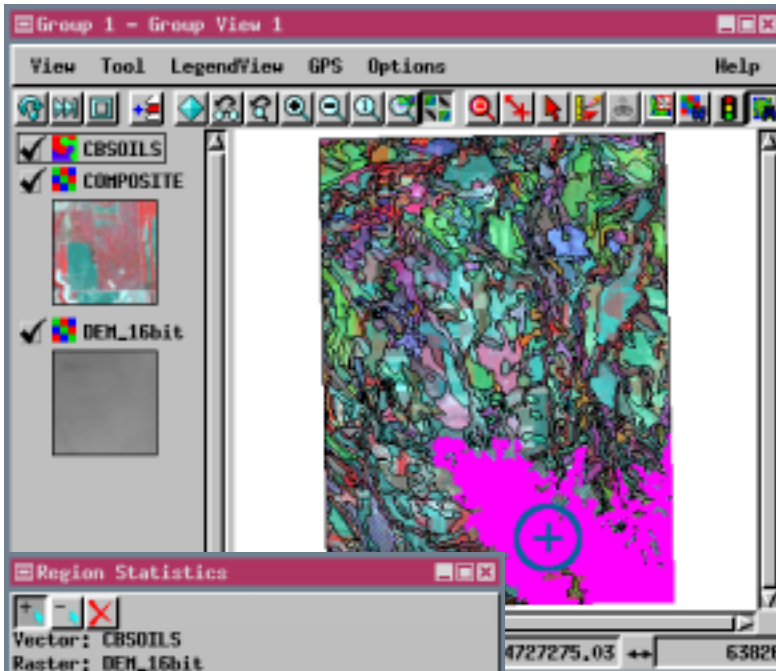
creates button row

creates standard polygon drawing tool

destroys the tool when necessary

Sample SML Tool Script

Region Statistics

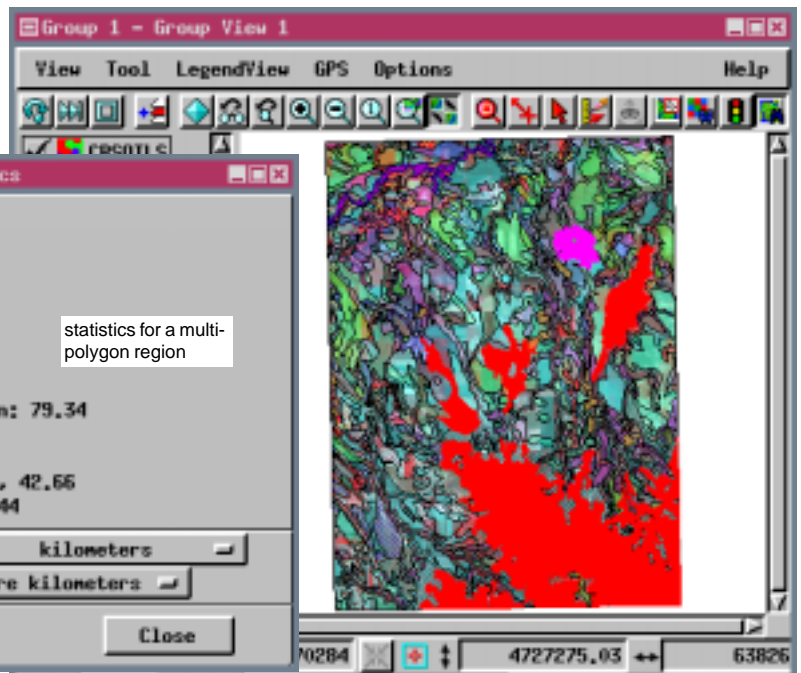
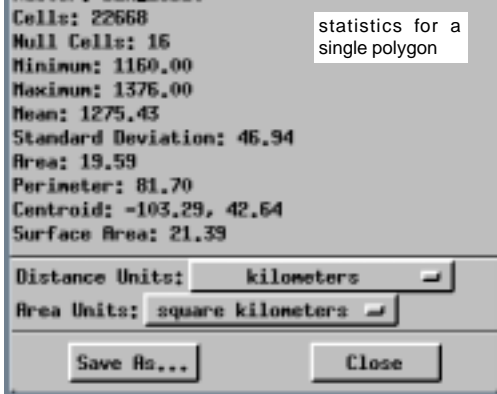


statistics for a single polygon

The Region Statistics tool script demonstrates how you can design a custom tool to visually select polygons and convert them to a temporary region to define the area for action on another coregistered layer. A tool with these functions could then use the region in a variety of operations. In this example, the region is used in the simple operation of extracting statistics from a raster object. This same tool could be modified to perform many other functions with the regions it creates. For example, it could extract points, lines, or polygons from another vector layer or use the extract functions to create rasters of the regions.

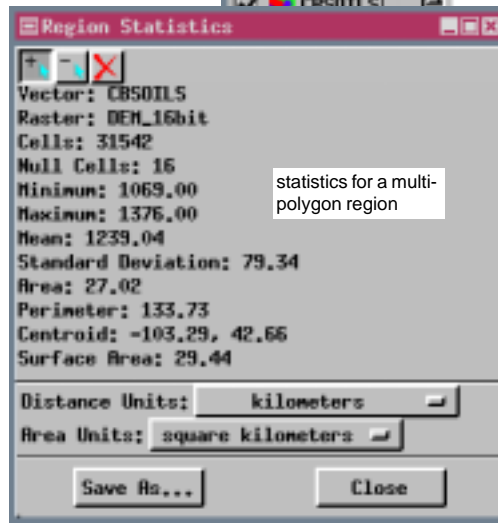
The Region Statistics script lets you select one or more vector polygons then calculates a set of statistics from an underlying raster for the area covered by the selected polygon(s). The top vector layer is used for polygon selection and the statistics are calculated for the bottom raster layer. The statistics calculated include the number of cells, the number of null cells, the minimum and

maximum cell values, the mean cell value, the standard deviation of cell values, the area of the region, its perimeter length, the coordinates of the centroid, and a surface area estimation. You can choose from any of the 25 length units and 13 area units standardly available throughout the TNT products for viewing the perimeter and area statistics. You can also save the calculated statistics as a text file. If you select the same text file again, the current statistics will be appended to earlier entries.



statistics for a multi-polygon region

This custom script reports the same statistics as the Area Statistics tool script. The method of area definition differs between the two—one has you draw the area and the other uses selected polygons to create a region. Together these scripts provide an excellent example of how to use the sample scripts provided by MicroImages to put together the pieces you need for your own custom tool. Think of the sample macro and tool scripts provided as a series of modules to be reused for the same or different purposes in a script that creates the custom tool you want. This script, for example, has modules concerned with selecting polygons, making a region from selected polygons, using a status window, calculating standard raster statistics for a local area, and estimating surface area from an elevation raster.



Macro and Tool Scripts can be created using SML in any TNTmips process that uses a View window (Options / Customize from the View window menu bar). These scripts are then available from an icon, which you select or design, on the toolbar. Sample scripts have been prepared to illustrate how you might use these features, which are available only in TNTmips 6.4 or later, to assist with specific tasks you perform on a regular basis. If possible, the full script is printed below for your quick perusal. When a script is too long to fit on one page, key sections are reproduced below. The sample Tool Script illustrated can be downloaded from the SML script exchange at www.microimages.com/sml/ftpsmlink/TNT_Products_V6.4_CD.

Partial Script for Region Statistics (regstatp.sml)

```
func checkLayer() {
  local boolean valid = false;

  if (Group.LastLayer.Type == "Vector") {
    vectorLayer = Group.LastLayer;
    DispGetVectorFromLayer(targetVector, vectorLayer);
    if (targetVector.SInfo.NumPolys > 0) {
      vectorName$ = vectorLayer.Name;
      valid = true;
    }
    else vectorName$ = "No polygons!";
  }
  else vectorName$ = "Not a vector!";
  if (Group.FirstLayer.Type == "Raster") {
    rasterLayer = Group.FirstLayer;
    DispGetRasterFromLayer(targetRaster, rasterLayer);
    if (targetRaster.SInfo.Type != "binary" and
        targetRaster.SInfo.Type != "4-bit unsigned" and
        targetRaster.SInfo.Type != "8-bit signed" and
        targetRaster.SInfo.Type != "8-bit unsigned" and
        targetRaster.SInfo.Type != "16-bit signed" and
        targetRaster.SInfo.Type != "16-bit unsigned" and
        targetRaster.SInfo.Type != "32-bit signed" and
        targetRaster.SInfo.Type != "32-bit unsigned" and
        targetRaster.SInfo.Type != "32-bit float" and
        targetRaster.SInfo.Type != "64-bit signed" and
        targetRaster.SInfo.Type != "64-bit unsigned" and
        targetRaster.SInfo.Type != "64-bit float") {
      rasterName$ = "Type not supported!";
      valid = false;
    }
    else
      rasterName$ = rasterLayer.Name;
  }
  else {
    rasterName$ = "Not a raster!";
    valid = false;
  }
  return valid;
}

proc cbRedraw() {
  local numeric larea, lperimeter, lsurface;

  larea = areaScale * area;
  lperimeter = distScale * perimeter;
  lsurface = areaScale * surface;
  if (gc == 0) return;
  ActivateGC(gc);

  SetColorName("gray75");
  FillRect(0, 0, da.width, da.height);
  SetColorName("black");
  if (cells > 0) {
    DrawInterfaceText(sprintf("Vector: %s\nRaster: %s\nCells: %d\nNull Cells: %d\nMinimum:
% .2f\nMaximum: % .2f\nMean: % .2f\nStandard Deviation: % .2f\nArea: % .2f\nPerimeter:
% .2f\nCentroid: % .2f, % .2f\nSurface Area: % .2f",
vectorName$, rasterName$, count, cells - count, min, max, mean, stdDev, larea, lperimeter,
centroid.x, centroid.y, lsurface), 0, 10);
  }
  else DrawInterfaceText(sprintf("Vector: %s\nRaster: %s\nCells:\nNull
Cells:\nMinimum:\nMaximum:\nMean:\nStandard Deviation:\nArea:\nPerimeter:\nCentroid:\nSurface
Area:", vectorName$, rasterName$), 0, 10);
}

proc cbToolApply(class pointTool tool) {
  if (checkLayer()) {
    local numeric sum, sumsqr, xscale, yscale, zscale;
    local numeric current, right, down, downright, elemNum;
    local region MyRgn;
    local class POINT2D point;
    local class StatusHandle status;
    local class StatusContext context;
    cells = 0; min = 0; max = 0; mean = 0; stdDev = 0; sum = 0; sumsqr = 0; count = 0; surface = 0;
    area = 0; perimeter = 0;
    centroid.x = 0; centroid.y = 0; current = 0; right = 0; down = 0; downright = 0;
    xscale = ColScale(targetRaster);
    yscale = LinScale(targetRaster);
    zscale = Group.FirstLayer.zscale;

```

checks to see if layers are valid

gets layer name if top layer is vector, gives error if not

gets layer name if bottom layer is valid raster type, gives error if not

scales the values to specified units

redraws Region Statistics window when new statistics computed

procedures after right mouse button click (tool applied)

defines local variables for statistics calculations

point.x = tool.Point.x;
point.y = tool.Point.y;

gets point coordinates of tool

```
point = TransPoint2D(point, ViewGetTransViewToScreen(View, 1));
point = TransPoint2D(point, ViewGetTransMapToView(View, vectorLayer.Projection, 1));

elemNum = FindClosestPoly(targetVector, point.x, point.y, GetLastUsedGeorefObject(targetVector));
```

```
if (elemNum > 0) {
  MyRgn = ConvertVectorPolyToRegion(targetVector, elemNum,
  GetLastUsedGeorefObject(targetVector));
  if (regionMode$ == "plus") {
    if (numRegions > 0) {
      MyRgn = RegionOR(reg, MyRgn);
    }
    numRegions += 1;
    vectorLayer.Poly.HighlightSingle(elemNum, 2);
  }
  else {
    if (numRegions > 1) {
      MyRgn = RegionSubtract(reg, MyRgn);
      numRegions -= 1;
    }
    else {
      MyRgn = RegionXOR(MyRgn, MyRgn);
      numRegions = 0;
    }
    vectorLayer.Poly.HighlightSingle(elemNum, 3);
  }
  reg = CopyRegion(MyRgn);

```

convert selected polygon(s) to region and transform to appropriate coordinate system

```
status = StatusDialogCreate(form);
context = StatusContextCreate(status);
StatusSetMessage(context, "Computing values...");
```

creates status dialog

```
foreach targetRaster[lin, col] in reg {
  if (!IsNull(targetRaster)) {
    if (count == 0) {
      max = targetRaster;
      min = targetRaster;
    }
    else if (targetRaster > max) {
      max = targetRaster;
    }
    else if (targetRaster < min) {
      min = targetRaster;
    }
    sum += targetRaster;
    sumsqr += sqr(targetRaster);
    count += 1;
  }

```

calculates statistics for the region

```
if (!IsNull(targetRaster))
  current = targetRaster;
if (!IsNull(targetRaster[lin,col+1]))
  right = targetRaster[lin,col+1];
if (!IsNull(targetRaster[lin+1,col]))
  down = targetRaster[lin+1,col];
if (!IsNull(targetRaster[lin+1,col+1]))
  downright = targetRaster[lin+1,col+1];
surface += .5*sqrt((sqr(yscale*current*zscale-yscale*right*zscale)
+sqr(xscale*current*zscale-xscale*down*zscale)
+sqr(xscale*yscale));
surface += .5*sqrt((sqr(yscale*downright*zscale-yscale*right*zscale)
+sqr(xscale*downright*zscale-xscale*down*zscale)
+sqr(xscale*yscale));

```

method for estimating surface area

```
cells += 1;
}
CloseRaster(targetRaster);
```

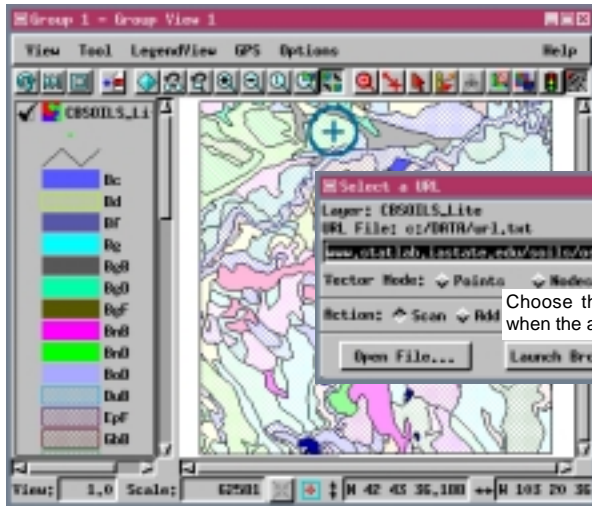
```
if (count > 1) {
  mean = sum / count;
  stdDev = sqrt((sumsqr - (sum^2 / count) / (count - 1)));
  area = reg.$Data.GetArea();
  perimeter = reg.$Data.GetPerimeter();
  centroid = reg.$Data.GetCentroid();
}
```

additional statistics calculated while avoiding division by zero

(see regstatp.sml for full script)

Sample SML Tool Script

Run Browser



The Run Browser script provides an example of a custom script used to launch an external application. The web browser was chosen as the example program because it is the one type of external program that all clients are most likely to have so you can run the script. The function that launches the external application is the same for any file type; it simply uses the file name provided to determine

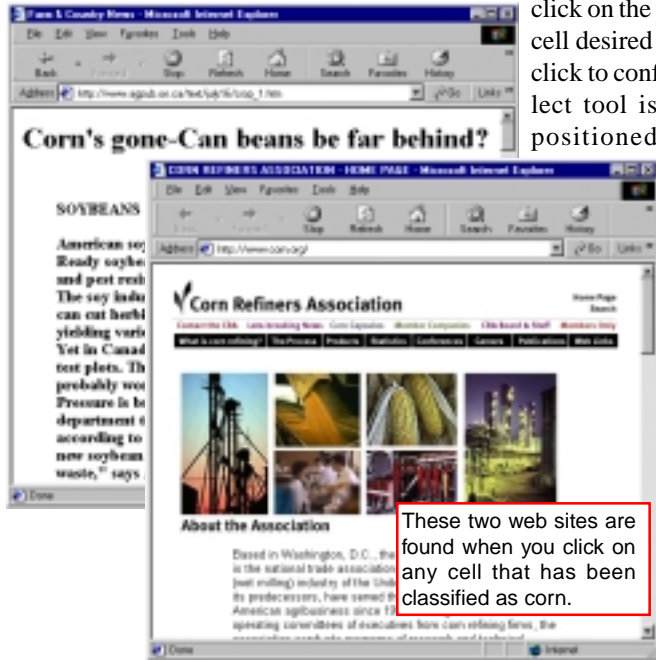
which application to launch. For example, file names that end in .ppt will launch Microsoft PowerPoint.

The Run Browser script lets you select elements in a vector or cell values in a raster layer, choose an associated URL for a web site to visit, then launch your Internet browser and go to the selected site. The associations between element attributes or cell values and URLs are specified in a separate text file, which means there are two files needed to use this tool (urls.sml and url.txt). The text file specifically lists the name and description for each object with URL links. Any number of objects can be listed in the file, but if the active layer in the group or layout does not contain one of the objects named, you get a "File not found" message instead of a list of URLs. To get results when you use this tool without altering the sample text file, you need to have either CBSOILS_LITE from the CB_SOILS Project File (CB_DATA folder) or CLS_MAXLIKE from the BERCRPCL Project File (BEREA folder) as the active layer. Both Project Files are found with the litedata on your TNT products CD-ROM or with the TNTlite datasets on MicroImages' web site.

You can associate one or more URLs with each attribute or cell value. You can also associate the same URL with different attributes if desired. The sample text file associates polygons of soil

type KeB, JmD, or Bd in CBSOILS_LITE with pages about the corresponding soil type at Iowa State University's web site. You could, of course, associate all the soil types with appropriate pages. To use the tool, left-

click on the polygon or cell desired then right-click to confirm the select tool is correctly positioned. The



These two web sites are found when you click on any cell that has been classified as corn.

URL(s) associated with where you clicked appear in the Select a URL window. Choose the desired URL, then click on the Launch Browser button. Your Internet browser will open if not open already and go to the designated web site and specified page.

You can easily add URL links for your own objects to this file. Toward the bottom of the Select a URL window there is an Action panel that lets you choose between Scan and Add. Switch to Add, and for a vector, left-click and right-click on an element of the type you want to link to. You are then prompted to select a table and field for the attribute and finally, to enter the URL you want elements with the selected attribute value to link to. For a raster, you are simply prompted to enter the URL to add for the cell value you clicked on. The necessary text with the proper syntax is entered in the url.txt file. You can then toggle on Scan for the action, select the cell or polygon, and go to the designated web site.

Macro and Tool Scripts can be created using SML in any TNTmips process that uses a View window (Options / Customize from the View window menu bar). These scripts are then available from an icon, which you select or design, on the toolbar. Sample scripts have been prepared to illustrate how you might use these features, which are available only in TNTmips 6.4 or later, to assist with specific tasks you perform on a regular basis. If possible, the full script is printed below for your quick perusal. When a script is too long to fit on one page, key sections are reproduced below. The sample Tool Script illustrated can be downloaded from the SML script exchange at www.microimages.com/sml/ftpsmlink/TNT_Products_V6.4_CD.

Reference File for Run Browser Script (url.txt)

```
[CBSOILS_Lite : Crow Butte soil type polygon overlay]
[poly CLASS Class KeB]
www.statlab.iastate.edu/soils/osd/dat/K/KEITH.html
[poly CLASS Class JmD]
www.statlab.iastate.edu/soils/osd/dat/J/JAYEM.html
[poly CLASS Class Bd]
www.statlab.iastate.edu/soils/osd/dat/B/BANKARD.html
```

```
[CLS_MAXLIKE : Class raster from 6_06, 7_30, & 10_10 (Green, Red, NIR6)]
{3}
```

object name
and description

URL

element type
and attribute
value specified

```
www.microimages.com
www.wheatworld.org
www.oznet.ksu.edu/wheatpage/
```

URLs linked
to cell value 3

```
{7}
{5}
www.agpub.on.ca/text/july16/crop_1.htm
```

cell values linked to
following URL(s)

```
{7}
www.corn.org
{5}
www.ag.uiuc.edu/~food-lab/soy/soy.html
```

Partial Script for Run Browser (urls.sml)

```
proc cbLayer() {
  if (Group.ActiveLayer.Type == "Raster") {
    vectorLabel.Sensitive = 0;
    pointButton.Sensitive = 0;
    nodeButton.Sensitive = 0;
    lineButton.Sensitive = 0;
    polyButton.Sensitive = 0
  }
  else {
    vectorLabel.Sensitive = 1;
    polyButton.Sensitive = 1;
    lineButton.Sensitive = 1;
    nodeButton.Sensitive = 1;
    pointButton.Sensitive = 1;
  }
  cbRedraw();
}
```

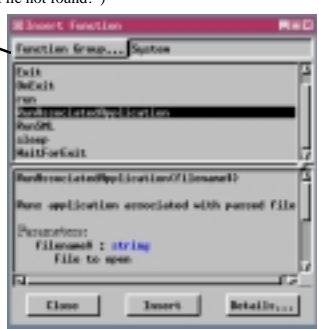
sets window options
when active layer
changes

```
proc cbOpen() {
  filepath$ = GetInputFileName(filepath$, "Open URL file", "txt");
  cbRedraw();
}
```

Open File button
functions

```
proc cbGo() {
  local string url$;
  url$ = list.GetItemAtPos(list.GetFirstSelectedPos());
  if (list.SelectedItemCount > 0 and url$ != "No URLs found!" and url$ != "Type not supported!"
  and url$ != "No element found!" and url$ != "File not found!")
    RunAssociatedApplication(url$);
}
```

Launch Browser
button functions



The RunAssociated Application function launches whatever application would be used if you double-clicked on the file on your desktop. If the file name ends in .doc, it will launch Microsoft Word; if the file name ends in .pdf, it will launch Adobe Acrobat or Acrobat Reader, whichever would be used if you double-clicked on the file on your desktop.

```
proc clearHighlight() {
  if (Group.ActiveLayer.Type == "Vector") {
    local class VECTORLAYER vl;
    vl = Group.ActiveLayer;
    if (mode$ == "point") {
      vl.Point.HighlightSingle(1);
      vl.Point.HighlightSingle(1, 3);
    }
    else if (mode$ == "node") {
      vl.Node.HighlightSingle(1);
      vl.Node.HighlightSingle(1, 3);
    }
    else if (mode$ == "line") {
      vl.Line.HighlightSingle(1);
      vl.Line.HighlightSingle(1, 3);
    }
    else {
      vl.Poly.HighlightSingle(1);
      vl.Poly.HighlightSingle(1, 3);
    }
  }
}
```

unhighlights
selected
element if
active layer
is vector

```
proc cbClose() {
  pointTool.Managed = 0;
  DialogClose(form);
  if (setDefaultWhenClose) {
    setDefaultWhenClose = false;
    View.SetDefaultTool();
  }
}
```

Close button
functions

```
proc cbModeChanged() {
  if (pointButton.Set == 1) {
    mode$ = "point";
  }
  else if (nodeButton.Set == 1) {
    mode$ = "node";
  }
}
```

changes vector
selection mode

```
else if (lineButton.Set == 1) {
  mode$ = "line";
}
else mode$ = "poly";
}
proc cbActionChanged() {
  if (scanButton.Set == 1) {
    action$ = "scan";
  }
  else action$ = "add";
}
```

sets Action
mode

```
proc cbToolApply(class RegionTool polyTool) {
  list.DeleteAllItems();
```

right mouse button click to confirm
selection does the following

clears the list

```
local string url$, layerName$, temp$, temp2$, item$, element$, table$, field$, value$;
local class FILE reffile;
local class LAYER layer;
local numeric numTok, i, j, num, start;
local class POINT2D point;
local class StatusHandle status;
local class StatusContext context;
layer = Group.ActiveLayer;
```

sets up local
variables

```
if (layer.Type == "Raster" or layer.Type == "Vector") {
  point.x = pointTool.Point.x;
  point.y = pointTool.Point.y;
```

keep going if active
layer is raster or vector

get coordinates of selected point

```
# Set up layer, object, layer name, and point transformations.
if (layer.Type == "Raster") {
  point = TransPoint2D(point, View.GetTransLayerToScreen(View, layer, 1));
  DispGetRasterFromLayer(rv, layer);
  layerName$ = "[" + rv.$Info.Name + " : " + rv.$Info.Desc + "];"
}
else if (layer.Type == "Vector") {
  point = TransPoint2D(point, View.GetTransViewToScreen(View, 1));
  point = TransPoint2D(point, View.GetTransMapToView(View, layer.Projection, 1));
  local class VECTORLAYER vl;
  vl = layer;
  DispGetVectorFromLayer(vv, layer);
  layerName$ = "[" + vv.$Info.Name + " : " + vv.$Info.Desc + "];"
}
reffile = fopen(filepath$);
```

determine that active layer
is named in reference file

```
start = 0;
while (!feof(reffile)) {
  url$ = fgetline$(reffile);
  start += 1;
  if (url$ == layerName$)
    break;
}
if (url$ != layerName$)
  list.AddItem("File not found!");
```

retrieve URL choices
for the active layer

```
url$ = "";
while (!feof(reffile)) {
  temp$ = fgetline$(reffile);
  url$ = url$ + temp$ + "\n";
  if (temp$ == "")
    break;
}
```

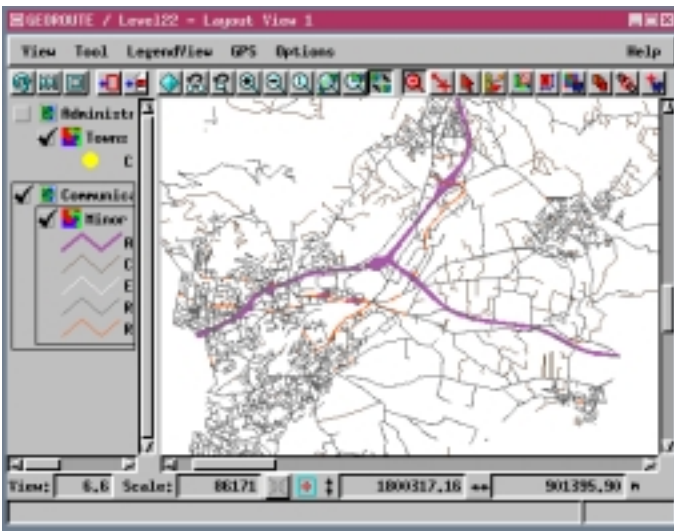
close the
reference file

```
fclose(reffile);
```

(see urls.sml for full script)

Sample SML Tool Script

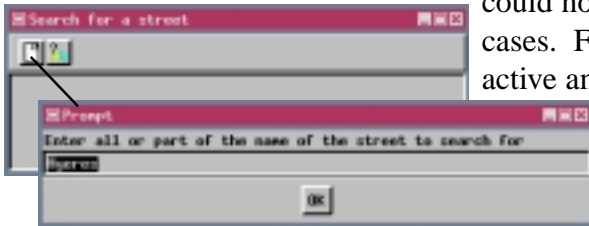
Find Streets



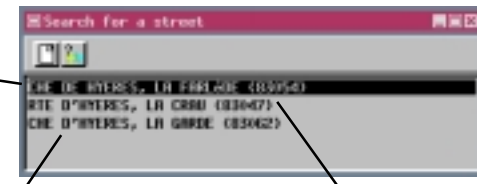
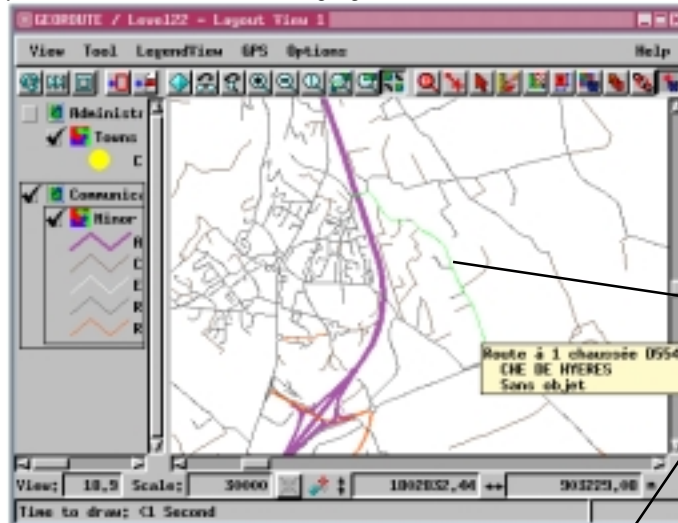
The Find Streets tool script locates and highlights streets you ask it to search for. You enter all or part of the street name to search for, and the tool script produces a list of all streets containing the text you entered. You then select the street you want to find and the script redraws the view at 1:30000 with the lines that form the street highlighted and centered in the View window. If all selected lines will not fit in the View at 1:30000, the View is redrawn at a smaller scale that fully contains the lines.

All lines that form the street are highlighted. The highlight colors for these lines are your designated selected and active element colors (Options / Colors). You may need to change these colors to take into account the drawing styles of the vector lines, which are purple, red, or black in this vector object. Thus, the default red highlight color could not be distinguished from unhighlighted lines in some cases. For the purposes of this tool, you may want to set the active and selected colors to be the same so that the selected street has a uniform appearance. The highlight color for the illustrations on this page is green.

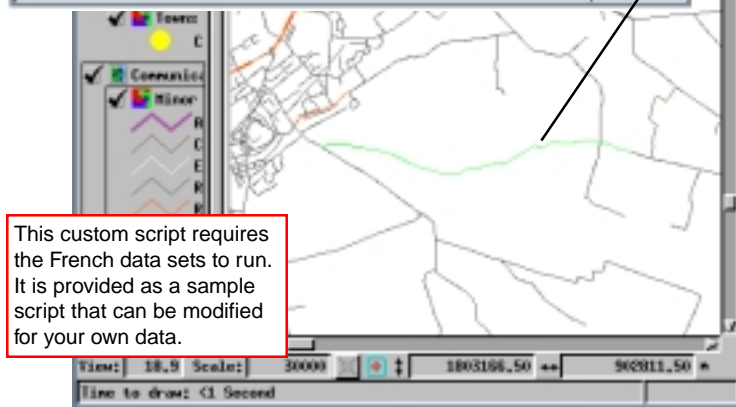
The New Search button opens a Prompt window so you can type in all or part of the name of the street you want to locate. Click OK, and all streets that match the search criteria are listed. Double-click on the name of the street you want to find or click on the *Highlight the street* icon.



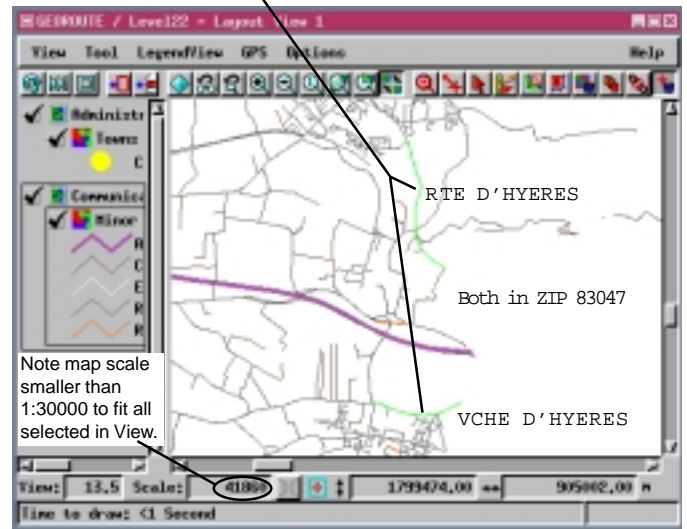
The name of the town and the zip code are also provided in the list of streets found. The script assumes there are not two separate streets in the same zip code with the same name. If, however, it turns out that the part of the street name you entered belongs to two different streets in the same zip code (one Main Street, the other Main Drive, for example), only the first encountered is listed but both are highlighted when that selection is made.



The script automatically pans to the selected street when you double click on the listing or click on the *Highlight the street* icon. The list is cleared when you click on New Search.



This custom script requires the French data sets to run. It is provided as a sample script that can be modified for your own data.



Note map scale smaller than 1:30000 to fit all selected in View.

Macro and Tool Scripts can be created using SML in any TNTmips process that uses a View window (Options / Customize from the View window menu bar). These scripts are then available from an icon, which you select or design, on the toolbar. Sample scripts have been prepared to illustrate how you might use these features, which are available only in TNTmips 6.4 or later, to assist with specific tasks you perform on a regular basis. If possible, the full script is printed below for your quick perusal. When a script is too long to fit on one page, key sections are reproduced below. The sample Tool Script illustrated can be downloaded from the SML script exchange at www.microimages.com/sml/ftpsmlink/TNT_Products_V6.4_CD.

Partial Script for Find Streets (street.sml)

```
# Zoom to selected position
proc DoZoom () {
    #Finding the element numbers of this street (not sure that they are sorted)
    local selpos;
    if (nline == 0) return;

    selpos = poslist.GetFirstSelectedPos();
    array streetline[1];
    nstreetline = 0;

    for i=1 to nline {
        curcode = V.line[linelist[i]].TRONCON_ROUTE.INSEE_COMD;
        if (curcode == codelist[selpos,1]) {
            nstreetline = nstreetline+1;
            ResizeArrayPreserve(streetline, nstreetline);
            streetline[nstreetline] = linelist[i];
        }
    }
    ViewSetMessage(View, NumToStr(nstreetline) + " lines found for this street");

    #Zoom in to the lines
    class VECTORLAYERLINES vll;
    vll = layer.Line;
    vll.HighlightMultiple(nstreetline, streetline);
    View.DisableRedraw = 1;
    layer.ZoomToHighlighted();
    if (View.GetMapScale(View) < 30000) {
        View.SetMapScale(View, 30000);
    }
    View.DisableRedraw = 0;
    View.Redraw(View);
}#DoZoom

# New Request
proc DoNew () {
    poslist.DeleteAllItems();
    nline = 0;

    ncode = 0;

    #asking to enter the name of a street (or a word contained in it)
    street$ = PopupString("Enter all or part of the name of the street to search for", "");
    if (street$ == "") return;
    street$ = touppers$(street$);

    #looking for a line containing street$ in its NOM_RUE_D or NOM_RUE_G attributes of
    the TRONCON_ROUTE table
    for i=1 to NumVectorLines(V) {
        #class DATABASE DB = V.line[i].TRONCON_ROUTE;
        if (V.line[i].TRONCON_ROUTE.NOM_RUE_DS contains street$ or
            V.line[i].TRONCON_ROUTE.NOM_RUE_GS contains street$) {
            nline = nline+1;
            linelist[nline] = i;
        }#if

    }#i
    ViewSetMessage(View, NumToStr(nline) + " lines found");

    if (nline == 0) {
        #no element corresponding found
        PopupMessage("No streets found containing this word!");
        return;
    }

    #Some streets are found : find the different ones (by zip code)
    #Assertion : not 2 streets with the same name in a town
    #Limits : don't take into account the streets separating 2 towns (the right zip code
    INSEE_COMD and the left one INSEE_COMG are different)
    for i=1 to nline {
        found = false;
        curcode = V.line[linelist[i]].TRONCON_ROUTE.INSEE_COMD;

        j=1;
        while (!found and j<=ncode) { #looks if code already found
            if (codelist[j,1] == curcode) {
                found = true;
            }#if
            j = j+1;
        }#while

        if (!found) { #new street
            ncode = ncode+1;
            codelist[ncode,1] = curcode;
            codelist[ncode,2] = linelist[i];
        }#if

    }#i
}#DoNew

#Retrieve the table containing the names of the towns
array townnames[ncode]; #V.Town.point[townname[i]].ZONE_HABITAT.INSEE ==
    V.line[codelist[i,2]].TRONCON_ROUTE.INSEE_COMD;
npts = NumVectorPoints(VTown);
ResizeArrayClear(townnames, ncode);

for i=1 to ncode {
    curcode = V.line[codelist[i,2]].TRONCON_ROUTE.INSEE_COMD;
    j = 1;
    found = false;
    townnames[i] = 0; #init
    while (!found and j<=npts) {
        if (VTown.point[j].ZONE_HABITAT.INSEE == curcode) {
            townnames[i] = j;
            found = true;
        }
        j = j+1;
    }
    if (townnames[i] == 0) { #error
        PopupMessage("Town name corresponding to " +
            NumToStr(curcode) + " not found");
    }
}#i

for i = 1 to ncode {
    name$ = V.line[codelist[i,2]].TRONCON_ROUTE.NOM_RUE_DS;
    zip$ = "(" + NumToStr(codelist[i,1]) + ")";
    town$ = " " + street$ =
        touppers$(VTown.point[townnames[i]].ZONE_HABITAT.TOPONYMES);
    poslist.AddItem(name$ + town$ + zip$);
}
poslist.SelectPos(1);
}#DoNew

# Close the window, switching to default tool
proc DoClose () {
    if (setDefaultWhenClose) {
        setDefaultWhenClose = false;
        View.SetDefaultTool();
    }
}

# Called the first time the tool is activated.
func OnInitialize () {
    dlgform = CreateFormDialog("Search for a street", View.Form);
    WidgetAddCallback(dlgform.Shell.PopdownCallback, DoClose);
    dlgform.Width = 200;

    class PushButtonItem btnItemNew;
    class PushButtonItem btnItemZoom;
    btnItemNew = CreatePushButtonItem("New search", DoNew);
    btnItemNew.IconName = "new";
    btnItemZoom = CreatePushButtonItem("Highlight the street", DoZoom);
    btnItemZoom.IconName = "apply_query";
    btnItemClose = CreatePushButtonItem("Close", DoClose);
    btnItemClose.IconName = "delete";

    # Icon button rows are automatically attached to their parent form on the left
    # and right. The "right" widget is unattached by setting the attachment to itself.

    class XmRowColumn btnrowaction;
    btnrowaction = CreateIconButtonRow(dlgform, btnItemNew, btnItemZoom);
    btnrowaction.TopWidget = dlgform;
    btnrowaction.TopOffset = 4;
    btnrowaction.TopOffset = 4;
    btnrowaction.LeftWidget = btnrowaction;
    btnrowaction.LeftOffset = 8;
    btnrowaction.RightOffset = 4;

    class XmSeparator btnsep;
    btnsep = CreateHorizontalSeparator(dlgform);
    btnsep.TopWidget = btnrowaction;
    btnsep.TopOffset = 4;
    btnsep.LeftWidget = dlgform;
    btnsep.RightWidget = dlgform;

    poslist = CreateScrolledList(dlgform);
}#OnInitialize

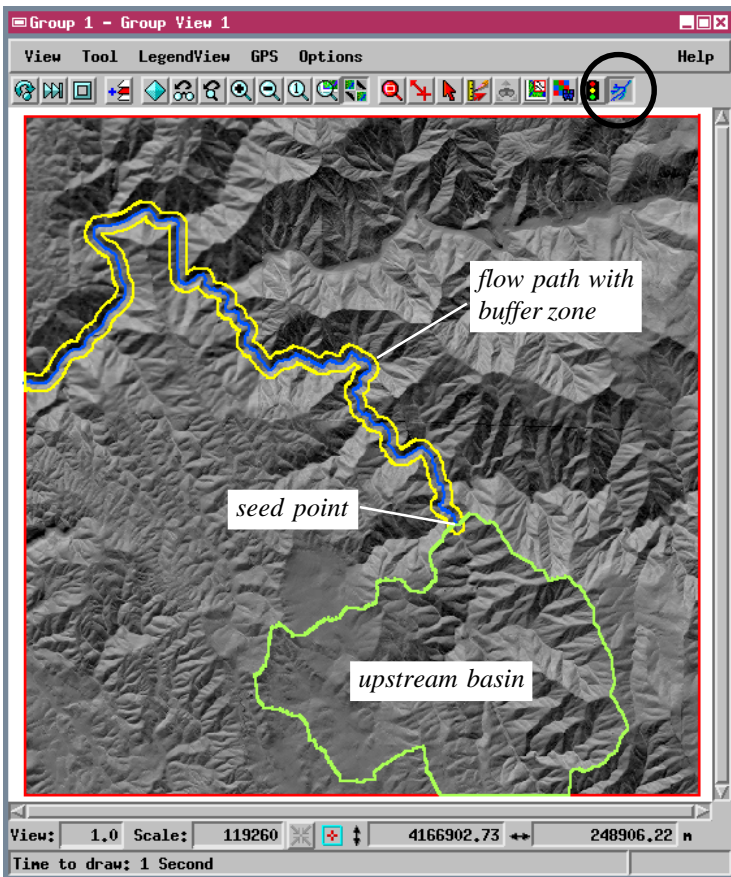
# creates search dialog
```

(see street.sml for full script)

Sample SML Tool Script Flow Path

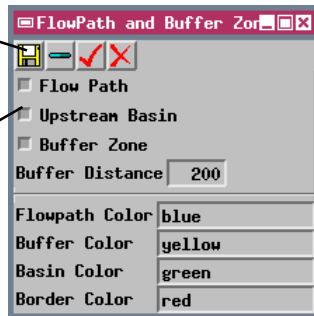
The Flow Path sample script shows how powerful custom analysis procedures can be performed on layers in the current view using an SML Tool Script. The script uses new SML watershed functions that operate on an elevation raster (DEM) in the View window. When you launch the script, it opens a FlowPath and Buffer Zone window and creates a graphic tool that allows you to place a watershed seed point on the DEM or on an overlying image layer. Depending on the options you have selected in the FlowPath window, the script computes and displays the upstream basin (area with flow toward the seed point), the flow path downstream, and a buffer zone around the flow path. You can move the seed point and repeat the analysis as many times as you like and save the computed elements at any time.

One application of this script is the evaluation of surface water pollution hazards. If the seed point represents a location where contamination has been detected, the upstream basin is the area of potential sources. If the seed point represents a contaminant spill, the flow path and buffer zone indicate the downstream area that is at risk.



Use the Save button to save the computed watershed features as vector objects in a Project File.

Use the toggle buttons to choose which watershed features you want computed and displayed.

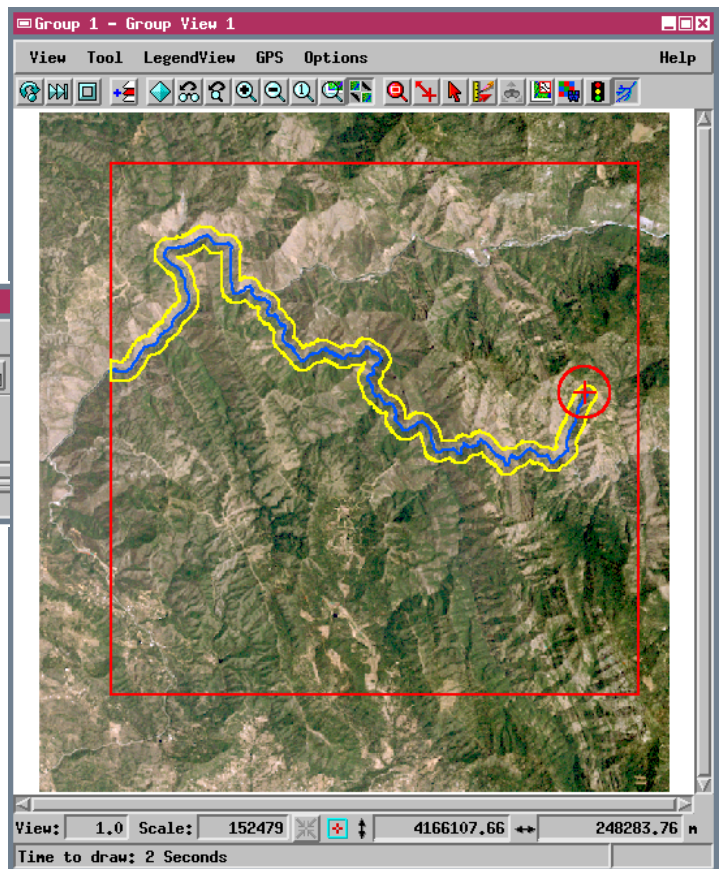


If the Display group has more than one layer, make sure that the elevation raster is the first layer in the group when you run the Flow Path Tool



Script. In the example to the right, an RGB raster layer showing a natural-color satellite image is displayed on top of the DEM. The extents of the DEM layer are computed by the script and displayed as a red rectangle. Use this rectangle as a guide to placing seed points when the overlying image extends beyond the DEM layer in one or more directions.

Development of this and other sample Tool Scripts continues at MicrolImages. Check the MicrolImages web site for an updated version incorporating additional features.



Macro and Tool Scripts can be created using SML in any TNTmips process that uses a View window (Options / Customize from the View window menu bar). These scripts are then available from an icon, which you select or design, on the toolbar. Sample scripts have been prepared to illustrate how you might use these features, which are available only in TNTmips 6.4 or later, to assist with specific tasks you perform on a regular basis. If possible, the full script is printed below for your quick perusal. When a script is too long to fit on one page, key sections are reproduced below. All sample Tool and Macro Scripts illustrated can be found in their entirety on your TNT products CD-ROM in the folder in which you installed TNTmips 6.4. These scripts, among others, can be downloaded from the SML script exchange at www.microimages.com/sml/ftpssmlink/TNT_Products_V6.4_CD.

Script for Flow Path (FlowPath.sm)

```

array seedx[10];
class WATERSHED w;
class RASTER DEM;
class POINT2D pt;
class VECTORLAYER VecBuf;
class VECTORLAYER VecFlow;
class VECTORLAYER BasinLayer;
class VECTORLAYER BoundaryLayer;
class VECTOR VecBoundary;
numeric xMax,xYMax,xMin,yMin;

array seedy[10];
numeric numpts;
numeric firstpass;
class PointTool point_tool;
class RasterLayer DEMLayer;
numeric haslayers;
class XmForm dlgform;
class VECTOR VectIn;
array xPoints[10],yPoints[10];
class PromptNum PromptDistance;

variable
declarations

btnsep.TopWidget = PromptDistance;
btnsep.TopOffset = 4;
btnsep.LeftWidget = dlgform;
btnsep.RightWidget = dlgform;

class PromptStr Promptflow;
class PromptStr Promptzone;
class PromptStr Promptbasin;
class PromptStr Promptborder;

Promptflow = CreatePromptStr(dlgform,"Flowpath Color",15,"blue");
Promptzone = CreatePromptStr(dlgform,"Buffer Color ",15,"yellow");
Promptbasin =CreatePromptStr(dlgform,"Basin Color ",15,"green");
Promptborder=CreatePromptStr(dlgform,"Border Color ",15,"red");

Promptflow.TopWidget = btnsep;
Promptflow.TopOffset = 4;
Promptflow.LeftWidget = dlgform;

Promptzone.TopWidget = Promptflow;
Promptzone.LeftWidget = dlgform;
Promptbasin.TopWidget = Promptzone;
Promptbasin.LeftWidget = dlgform;
Promptborder.TopWidget = Promptbasin;
Promptborder.LeftWidget = dlgform;
} # end of OnInitialize

func OnInitialize() {
if (Group.FirstLayer.Type == "Raster") {
DispGetRasterFromLayer (DEM,Group.FirstLayer);
DEMLayer = Group.FirstLayer;
}
else {
PopupString("First Layer must be a raster object for Watershed
Toolscript");
WaitForExit();
}

demFilename$ = GetObjectFileName(DEM);
demInode = GetObjectNumber(DEM);
demObjname$ = GetObjectName(demFilename$,demInode);

w = WatershedInit(demFilename$,demObjname$);

WatershedCompute(w,"FillAllDepressions");

firstpass = 1; haslayers = 0; numpts = 1;

dlgform = CreateFormDialog("FlowPath and Buffer Zone",View.Form);
WidgetAddCallback(dlgform.Shell.PopdownCallback,DoClose);

class PushButtonItem btnItemSave;
class PushButtonItem btnItemRemove;
class PushButtonItem btnItemSet;
class PushButtonItem btnItemClose;

btnItemSave = CreatePushButtonItem("Save Output Layers...",DoSave);
btnItemSave.IconName = "save";
btnItemRemove = CreatePushButtonItem("Remove Output Layers",
cbDoRemove);
btnItemRemove.IconName = "remove_sel";
btnItemSet = CreatePushButtonItem("Set Number of Seedpoints...",
DoSet);
btnItemSet.IconName = "apply";
btnItemClose = CreatePushButtonItem("Close",DoClose);
btnItemClose.IconName = "delete";

class XmRowColumn btnrowaction;
btnrowaction = CreateIconButtonRow(dlgform,btnItemSave,
btnItemRemove,btnItemSet,btnItemClose);
btnrowaction.TopWidget = dlgform;
btnrowaction.RightWidget = dlgform;
btnrowaction.LeftWidget = dlgform;

class XmToggleButton btnFlow;
class XmToggleButton btnBasin;
class XmToggleButton btnBuffer;

btnFlow = CreateToggleButton(dlgform,"Flow Path");
btnBasin = CreateToggleButton(dlgform,"Upstream Basin");
btnBuffer = CreateToggleButton(dlgform,"Buffer Zone");
PromptDistance = CreatePromptNum(dlgform,"Buffer Distance",
5,0,100);

btnFlow.Set = 1; btnBasin.Set = 1; btnBuffer.Set = 1;

btnFlow.TopWidget = btnrowaction;
btnFlow.LeftWidget = dlgform;
btnBasin.TopWidget = btnFlow;
btnBasin.LeftWidget = dlgform;
btnBuffer.TopWidget = btnBasin;
btnBuffer.LeftWidget = dlgform;
PromptDistance.TopWidget = btnBuffer;
PromptDistance.LeftWidget = dlgform;

class XmSeparator btnsep;
btnsep = CreateHorizontalSeparator(dlgform);

proc DoFlowPath() {
View.DisableRedraw = 1;
if ((btnFlow.Set == 0) and (btnBuffer.Set == 0) and
(btnBasin.Set == 0)) {
return;
}
if ((btnFlow.Set == 1) or (btnBuffer.Set == 1) and
(btnBasin.Set == 1)) {
WatershedComputeElements(w,seedx,seedy,numpts,"FlowPath,Basin");
}
if ((btnBasin.Set == 1) and (btnBuffer.Set == 0) and
(btnFlow.Set == 0)){
WatershedComputeElements(w,seedx,seedy,numpts,"Basin");
}
if (btnBasin.Set == 0) {
WatershedComputeElements(w,seedx,seedy,numpts,"FlowPath");
}

if ((btnFlow.Set == 1) or (btnBuffer.Set == 1)) {
WatershedGetObject(w,"VectorUserFlowPath",userflowpathFilename$,
userflowpathObjname$);
OpenVector(VectIn,userflowpathFilename$,userflowpathObjname$);
}

if (btnFlow.Set == 1) {
VecFlow = GroupQuickAddVectorVar(Group,VectIn);
VecFlow.Line.NormalStyle.Color.name = Promptflow.value;
}

if (btnBuffer.Set == 1) {
CreateTempVector(Buffer);
CreateTempVector(TempBuffer);
TempBuffer = VectorToBufferZone(VectIn,"line",
PromptDistance.value,"meters");
Buffer = VectorExtract(VecBoundary,TempBuffer,"InsideClip");
VecBuf = GroupQuickAddVectorVar(Group,Buffer);
VecBuf.Line.NormalStyle.Color.name = Promptzone.value;
}

if (btnBasin.Set == 1) {
WatershedGetObject(w,"VectorUserBasin",userBasinFilename$,
userBasinObjname$);
OpenVector(BasinVector,userBasinFilename$,userBasinObjname$);
BasinLayer = GroupQuickAddVectorVar(Group,BasinVector);
BasinLayer.Line.NormalStyle.Color.name = Promptbasin.value;
}

BoundaryLayer.Line.NormalStyle.Color.name = Promptborder.value;
View.DisableRedraw = 0;
View.RedrawIfNeeded(View);
haslayers = 1;
} # end of DoFlowPath

```

is called the first time the tool is activated

initializes watershed object (w); compute depressionless DEM

creates "FlowPath and Buffer Zone" dialog window

computes flow path, buffer zone, and basin originating at seed point (depending on options selected in dialog)

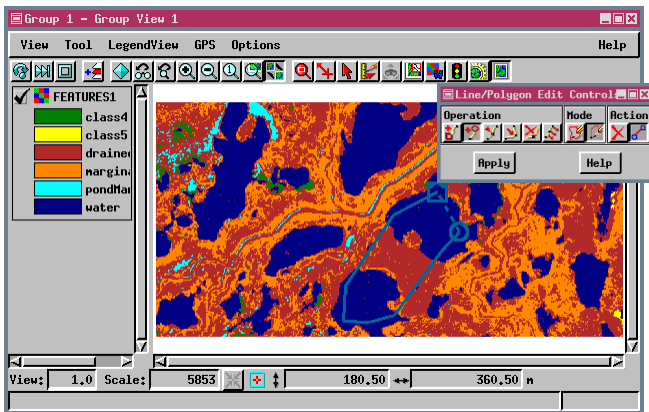
adds flow path vector to view

computes optional buffer zone around flow path and add to view

adds basin vector to view

Sample SML Tool Script

Running FRAGSTATS with TNTmips



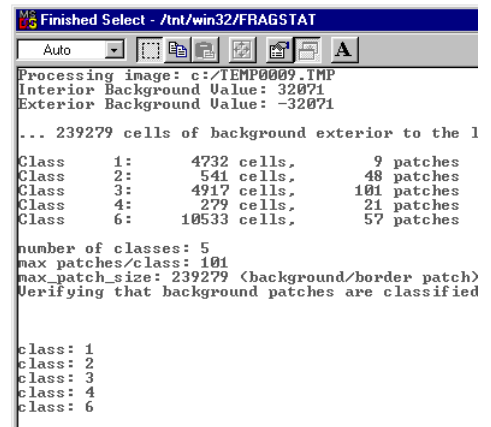
The patch, which is the basic unit of landscape ecology, is a piece of the landscape that is considered homogeneous at the scale of a particular study. A landscape, or area of interest, is a mosaic of patches of different types. Patch type is equivalent to a class in TNTmips terminology (patches of a single type in the landscape belong to the same class).

In order to study landscape function and change, you need to be able to quantify landscape structure. The FRAGSTATS program was developed for this purpose by Kevin McGarigal and Barbara Marks. FRAGSTATS calculates a number of statistics for each patch, for all the patches of a single class, and for the

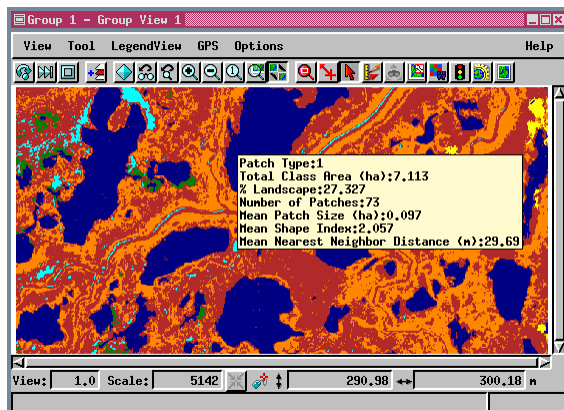
landscape as a whole. FRAGSTATS is concerned with both landscape composition and landscape configuration. Landscape composition addresses the variety and abundance of patches within the landscape, while landscape configuration is concerned with physical distribution and spatial character of patches.

FRAGSTATS runs under DOS and outputs four files, each with the name you provide and a different extension (*.cla, *.ful, *.lnd, *.pat). Because it runs under DOS, output file names are restricted to eight characters. Three of these files are designed for direct database import from text while the fourth file (*.ful) combines information on individual patches, patch classes, and the landscape as a whole into a single, more humanly readable report.

Two separate scripts for running FRAGSTATS are available with this release of the TNT products. One is a tool script that lets you draw a region to define the area of the underlying raster to use for calculation of statistics. The other script is run through the SML process and requires the landscape raster and a mask raster to define your area of interest within the landscape. The FRAGSTATS tool script demonstrates that a tool script can run an external program using objects from TNTmips Project Files and that FRAGSTATS can be used with an interactively designed mask (a region).



Once you have drawn and applied your region (or selected both the landscape and mask raster), you are asked to supply an edge distance in meters. The edge distance is the setback distance (in meters) within each patch for the purpose of calculating core area metrics. Next, a DOS shell opens and reports progress while FRAGSTATS is running. When FRAGSTATS is done, the DOS shell will say *finished* in the title bar. You need to close the DOS shell in order to continue working in TNTmips' Spatial Data Display process. The amount of time it takes FRAGSTATS to run is determined by the number of cells selected for processing and the number of patches within the selected area.



Summary statistics for each patch type were imported to a database related to the landscape raster by cell value. A multiline DataTip that incorporates specific statistics of interest (from the 40 in the *.cla file) was then constructed using a string expression field.

The information shown in the multiline DataTip (left) was selected from the table that resulted from import of the .cla file produced by FRAGSTATS. The imported table can be related to the internal table using cell value (Internal.Value), which corresponds to the patch type in the FRAGSTATS output.

Sample SML Tool Script

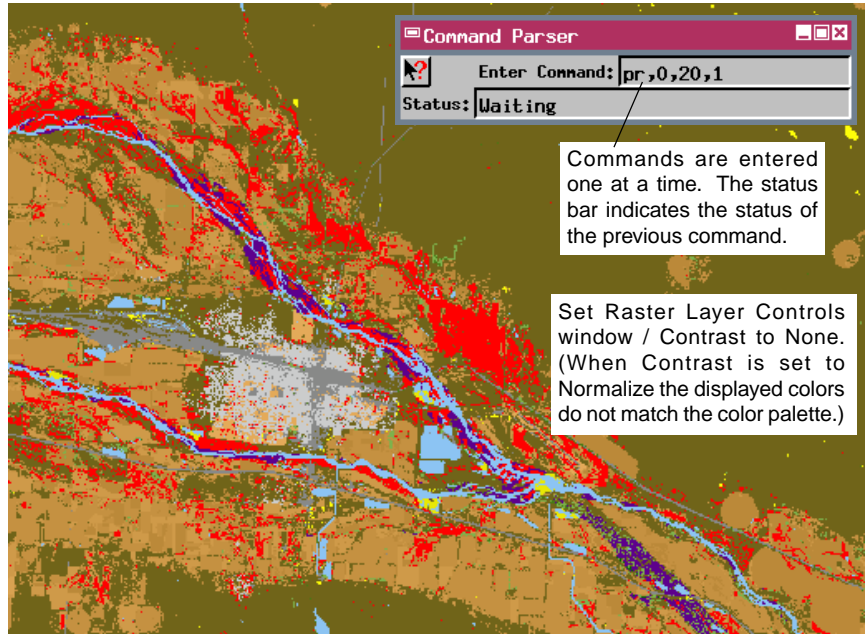
Printing Fixed Colors

The `compar.sml` script provides an example that can be used and modified to assign colors to cell values in an 8-bit raster in a quantitative fashion. If your printer is color stable, you can determine which numerically input RGB color value produces the exact color required. This script will then let you specify a cell value in your input raster and assign it a precise RGB color in the associated color palette. The script was written for someone who wanted a fast way to assign a specific color to each class in a classified image. Assuming your color printer does not drift in color, this approach will ensure that each class is printed in a consistent color from raster to raster. This script also illustrates how user input is parsed.

When you select this tool, a Command Parser window opens for you to enter commands.

The paint commands require you to specify a color with a color number. The script looks up the color number in a specified text file, which contains sequential color numbers (called index numbers) from 0 to 255. Each index number has red, green, and blue color component percentages as well as a transparency value. You can create this file from any raster's color palette by using the `t` command. Edit this file or even create it from scratch using a text editor or spreadsheet program. Depending on the command you enter, the script can copy these colors all at once to a raster's color palette (`lc` command) or load them (`b` command) if you want to access colors individually in order to paint specific raster cells (`p` and `pr` commands).

The best way to find the index number of a color (or a class represented by that color) is to make a standard color chart you can refer to. A color chart makes it easy to use any number of color shades to represent unique classes. All you have to do is refer to the chart, find the index number of the color you need, and use the `compar.sml` tool to paint all corresponding raster cells. The color chart is even more important when you have a large number of features to classify and you have no



choice but to use similar colors for different feature classes. Even with a few classes, subtle but noticeable changes in brightness and shade are an attractive and effective way to convey information about one class relative to another. There are many ways to create a color chart; the simplest is to list all of your color numbers along with the feature classes they correspond to. The chart to the left was created in TNTmips.

Before painting a raster, enter the `b` command to load the text file that contains the index numbers and color component percentages for all your colors. Use the `p` and `pr` commands to paint the raster. All you need to know to use these commands is the color number corresponding to the class, which you can get from your color chart, and the corresponding cell values that you want to paint.

Group 1 - Group View 1			
View Tool LegendView GPS Options			
Woody Herbaceous Wetlands	0	Emergent Residential Wetlands	1
Herbaceous planted/cultivated urban/recreational Grasses	5	Herbaceous Fallow	6
Shrubland	10	Orchards/ Vineyard	11
Deciduous Forest	15	Evergreen Forest	16
Barren Bare Rock	20	Barren Quarries	21
		Barren Transitional	22
		Low Intensity Residential	2
		High Int Reside	3
		Herbaceous Small Grains	7
		Herbaceous Pasture Hay	8
		Grasslands	12
		Water	13

CLASS: Emergent
Herbaceous Wetlands
Color # (index): 1
Red: 100%
Green: 0%
Blue: 0%
Transparent: 0%

Commands with sample variables:

- `t` (outputs color palette to text file)
- `lc` (make a new SMLcolor palette from text file)
- `b` (load text file colors to use to paint)
- `r,0,255` (cell value 0 to 255 becomes transparent)
- `pr,0,20,1` (paints range of cell values 0 to 20 with color # 1)
- `p,21,2` (paint cell value 21 with color # 2)

Sample scripts have been prepared to illustrate how you might use the features of TNTmips' Spatial Manipulation Language (SML). Key sections of the script are printed below for your quick perusal. The entire script can be downloaded from the SML script exchange at www.microimages.com/sml/ftpsmlink/TNT_Products_V6.5_CD.

Excerpts from Command Parser Tool Script (compar.sml)

```

proc OutputColorMap() {
  if (Group.ActiveLayer.Type == "Raster") {
    DispGetRasterFromLayer(rast,Group.ActiveLayer);
    cmap = ColorMapFromRastVar(rast);
    if (cmap.name == "") {
      PopupMessage("No ColorMap was found in raster object Paint function
      aborted\n Use lc command to save a csv file as a colormap in the raster
      object first\n or create one yourself");
      Command.value = "";
      Status.value = "OutputColorMap aborted";
      return;
    }
    myfile = GetOutputTextFile("c:/color.csv","Select file for output:","csv");
    fprintf(myfile,"%s,%s,%s,%s,%s\n","Index","Red","Green","Blue",
      "Transparency");
    Status.value = "Outputting Colormap";
    for i = 0 to 255 {
      mycolor = ColorMapGetColor(cmap,i);
      fprintf(myfile,"%d,%d,%d,%d,%d\n",i,round(mycolor.red),
        round(mycolor.green),round(mycolor.blue),
        round(mycolor.transp));
    }
    fclose(myfile);
    Status.value = "Colormap outputted";
    Command.value = "";
  }
  else {
    Status.value = "Active Layer must be a raster w/colormap for this function";
    Command.value = "";
  }
}

proc Paint() {
  if (NumberTokens(Command.value,delim) != 3) {
    Status.value = "Not enough parameters for Paint function";
    return;
  }
  if (!hascolorarray) {
    PopupMessage("Use code b (SetColorArray function) to load a colormap
    to use first");
    Status.value = "Paint function aborted";
    Command.value = "";
    return;
  }
  local numeric cellvalue;
  local numeric colormapnumber;
  cellvalue = StrToNum(GetToken(Command.value,delim,2));
  colormapnumber = StrToNum(GetToken(Command.value,delim,3));
  if (Group.ActiveLayer.Type != "Raster") {
    PopupMessage("Active Layer must be a raster object for this function");
    Command.value = "";
    Status.value = "Active layer must be a raster";
    return;
  }
  DispGetRasterFromLayer(rast,Group.ActiveLayer);
  cmap = ColorMapFromRastVar(rast);
  if (cmap.name == "") {
    PopupMessage("No ColorMap was found in raster object Paint function
    aborted\n Use lc command to save a csv file as a colormap in the raster
    object first\n or create one yourself");
    Command.value = "";
    Status.value = "Paint function aborted aborted";
    return;
  }
  mycolor.red = ared[colormapnumber+1];
  mycolor.green = agreen[colormapnumber+1];
  mycolor.blue = ablue[colormapnumber+1];
  mycolor.transp = atransp[colormapnumber+1];
  ColorMapSetColor(cmap,cellvalue,mycolor);
}

```

OutputColorMap called when t command entered; copies color map in Raster to CSV file
Command Syntax: t

Opens Select File window for you choose the output file name

Returns the color at index 'i' in color palette 'cmap'

fprintf output the CSV file, example:

Index	Red	Green	Blue	Transparency
0	0	0	0	50
1	0	100	0	50
2	0	100	100	50
.
.
255	100	100	100	100

Paint procedure invoked when p command entered; sets 'cellvalue' in color palette to 'index' value in array
Command Syntax: p,cellvalue,index

'hascolorarray' is initially set to false; when b command (SetColorArray function) is executed it is set to true

Gets Raster 'rast' from active layer

Returns Raster's color palette

array variables (ared, agreen, ablue, & atransp) hold paint values; 'colormapnumber' can be 0-255; you must add one since the array holds colors 1-256 (SML arrays start at 1)

Sets a color palette color
ColorMapSetColor(colormap,index,color)

```

ColorMapWriteToRastVar(rast,cmap,cmap.Name,cmap.Desc);

View.DisableRedraw = 1;
LayerDestroy(Group.ActiveLayer);
GroupQuickAddRasterVar(Group,rast);
Group.ActiveLayer.AllowDeleteLayer = 1;
View.DisableRedraw = 0;
ViewRedraw(View);

Status.value = "Cell Painted";
Command.value = "";
}

proc LoadColorMap() {
  local string line;
  myfile = GetInputTextFile("c:/colormap.csv","Select Colormap file","csv");
  Checks to see if the input file is valid:
  line = fgetline(myfile);
  if ((NumberTokens(line,",") != 5) || ("Index" != GetToken(line,",",1))) {
    PopupMessage("This does not appear to be a valid colormap csv file\n
    The proper format is one row of labels then 256 numeric lines of the
    form red,green,blue,transp\n where red,green,blue,transp are in the range
    0-100\n Display a raster with a colormap already and use the t command to
    export the colormap to a file to see how the file should look");
    Status.value = "Fatal Error Colormap load halted";
    Command.value = "";
    return;
  }
  Status.value = "Loading Colormap";
  for i = 0 to 255 {
    line = fgetline(myfile);
    Additional checks to see if the input file is valid:
    if ((i > 0) && (StrToNum(GetToken(line,",",1)) == 0)) {
      PopupMessage("Bad Index Value Encountered while loading Colormap file
      Colormap load aborted");
      Status.value = "Fatal Error Colormap load halted";
      Command.value = "";
      return;
    }
    mycolor.red = StrToNum(GetToken(line,",",2));
    mycolor.green = StrToNum(GetToken(line,",",3));
    mycolor.blue = StrToNum(GetToken(line,",",4));
    mycolor.transp = StrToNum(GetToken(line,",",5));
    ColorMapSetColor(cmap,i,mycolor);
  }
  if (Group.ActiveLayer.Type != "Raster") {
    PopupMessage("Active Layer must be a raster object for this function");
    Command.value = "";
    Status.value = "Active layer must be a raster";
    return;
  }
  DispGetRasterFromLayer(rast,Group.ActiveLayer);
  ColorMapWriteToRastVar(rast,cmap,"SMLcolor",
    ColorMap created by SML script");
  Status.value = "ColorMap saved as SMLcolor in raster object";
  PopupMessage("Colormap was saved as a SMLcolor colormap under raster
  object\n you must select this colormap to see the changes\n if you were displaying
  the raster with a different colormap\n this is the only function that behaves this
  way\n All other function modify the most recently used colormap");
  Command.value = "";
}

```

Writes a color palette under a raster
ColorMapWriteToRastVar(Raster,colormap.Name\$,description\$)

* These six lines of code force raster to reload with edited color palette

LoadColorMap procedure invoked when lc command is entered; creates a new color map for the Raster
Command Syntax: lc

Opens Select File window for you choose the input file name

Get next line of text from the csv file

GetToken(string\$,delimiters\$,tokenNumber) gets a token, which is a delimited portion of a string

writes new color palette called SMLcolor for Raster;
ColorMapWriteToRastVar(Raster, colormap, name\$, description\$)

code to force raster to reload with new color palette is not shown (*see identical lines of code in Paint procedure)

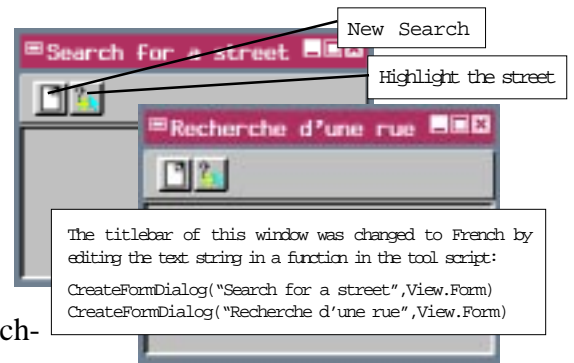
global variables declared outside of these procedures

class RASTER rast;	class FILE myfile;	array ablue[256];
class ColorMap cmap;	array ared[256];	array atransp[256];
class Color mycolor;	array agreen[256];	numeric hascolorarray;

Localizing SML Scripts

SML dialogs, including Macro and Tool Scripts, should conform to the same language as the rest of your interface. The text strings for dialog boxes and other interface components of any SML script can be customized to your own language.

In this example, an SML tool script dialog is changed from English to French. The new French version of the tool can then be made available in the sample atlas of France and have the same language as the rest of the dialog. The *Find Streets* tool (*street.sml*) lets the user enter a street name and then zooms in on and highlights any matching streets found in a specific vector attribute table. To update the language of all interface components, open the SML tool script in an editor and examine and change any functions that affect text in the interface. The ToolTips for the icon buttons in the *Search for a street* window shown above are specified by string parameters in two *CreatePushButtonItem* functions in the script. Any *Create* function should be examined and their text strings replaced. Some other functions with text strings to look for include: *PopupString* (like *PopupString* and *PopupMessage*), *print*, *fprint*, *sprint*, *SetStatusMessage*, *StatusSetMessage*, and *ViewSetMessage*.



Text strings that were changed in the *street.sml* tool script include:

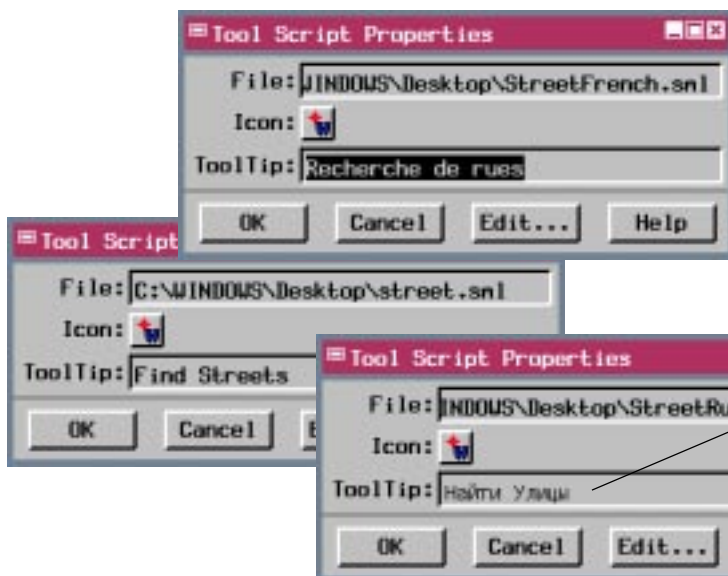
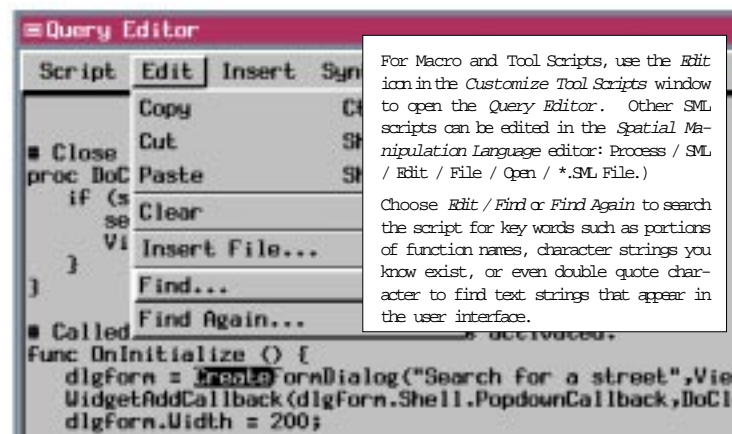
```
CreatePushButtonItem("New search",DoNew);
CreatePushButtonItem("Nouvelle requête",DoNew)

CreatePushButtonItem("Highlight the street",DoZoom)
CreatePushButtonItem("Afficher la rue",DoZoom)

CreateFormDialog("Search for a street",View.Form)
CreateFormDialog("Recherche d'une rue",View.Form)

PopupString("Enter all or part of the name of the
street to search for", "")
PopupString("Entrez le nom ou une partie du nom
de la rue a chercher", "")

PopupMessage("No streets found containing this word!")
PopupMessage("Aucune rue de l'échantillon de la base
de donnée ne contient ce mot!")
```



In addition to changing text strings in the script, you should also change the ToolTip for the macro or tool icon button in the View window. Open the *Customize Tool Scripts* window (*Options / Customize / Tool Scripts*), click the *Properties* icon and change the ToolTip.

For MS Windows operating systems, to change the character set for your computer keyboard, open the *Keyboard Properties* window (*Control Panel / Keyboard*) and add your language. Then switch your keyboard between character sets by typing the specified keyboard shortcut. (Make sure the environment you want to type in, Windows or ML/X, has focus before typing the shortcut to change languages.)