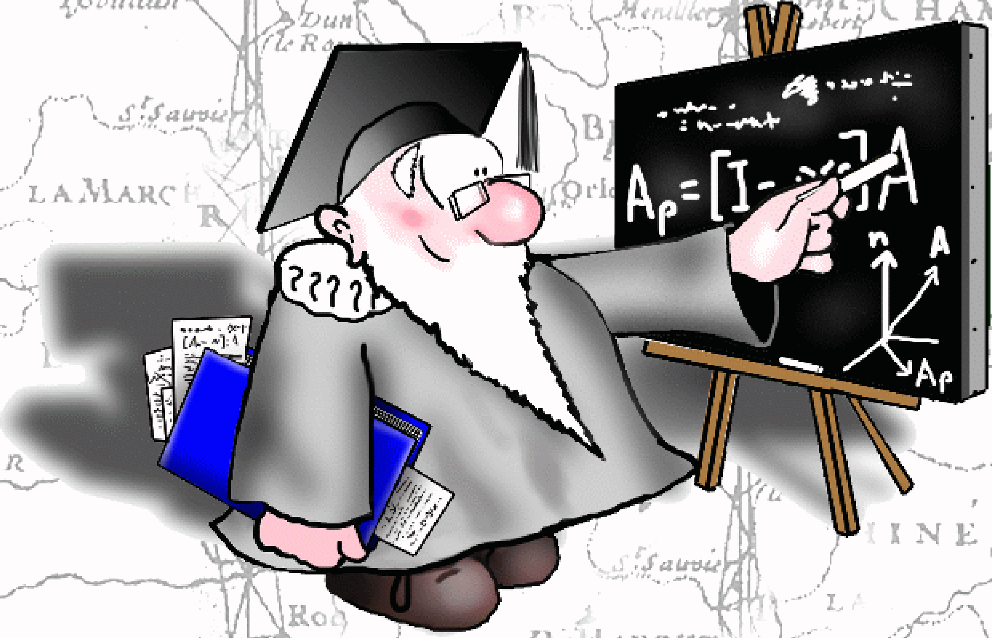
Tutorial

Writing Scripts with SML



in

TNTmips®
TNTedit™
TNTview®

Before Getting Started

This booklet introduces the fundamentals of creating scripts in the Spatial Manipulation Language (SML) in the TNT products. The exercises in this booklet introduce you to basic SML concepts and scripting conventions and provide many examples of 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 the *Displaying Geospatial Data* and *Navigating* tutorial booklets. 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 distributed with the TNT products in the DATA and SCRIPTS directories. If you do not have access to a TNT products CD, you can download the data from the MicroImages web site. This booklet uses scripts in the SML subdirectory of DATA, and in the MACRSCR and TOOLSCR subdirectories in the SCRIPTS directory. You will also need files in the CB_DATA, SF_DATA, SURFMODL, and EDITRAST subdirectories of DATA. Make a read-write copy of the sample data on your hard drive so changes can be saved when you use these objects.

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 the size of your project materials and does not allow export. All exercises in this booklet can be completed in TNTlite using the sample geodata provided.

Randall B. Smith, Ph.D. and Keith Ghormley, 15 October 2003

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 tutorial 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 the general-purpose scripting language used throughout the TNT products. If you have written a selection query, you have already used basic elements of SML. But you can also use SML to design custom processes and add unique capabilities to the TNT products. SML has evolved as the capabilities of the TNT products have grown. From its origins as a scripting language for custom processing of raster images, SML can now process any type of spatial object and associated database information. You can use SML scripts to operate on the geospatial data objects in Project Files, on objects displayed in a spatial view, or even to create a virtual display layer in a view.

You can create and use custom SML scripts in TNTmips, TNTedit, or TNTview. Scripts prepared for use in these products can also be distributed and used in TNTatlas and TNTatlas for Windows. SML scripts are platform-independent; they run without modification on any computer that runs TNTmips.

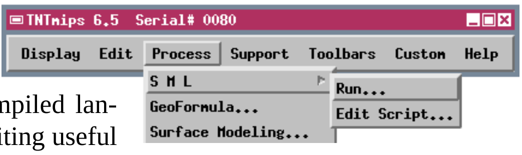
SML is an *interpreted* scripting language. This means that your computer evaluates and executes script statements one at a time. Interpreted languages are slower than *compiled* languages (like C or Pascal) in which the program code is pre-evaluated to create a fast, machine-readable version. On the other hand, SML has a simpler structure and syntax than compiled languages, making the task of writing useful scripts much easier. And SML provides access to many of the compiled functions and processes found in TNTmips, which can speed script execution for complex operations.

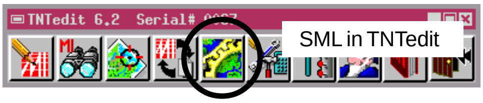
STEPS
☑ select Process / SML / Edit Script... from the TNTmips main menu

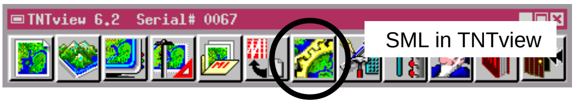
The exercises on pp. 4-18 introduce basic SML concepts and scripting conventions. Pages 19-27 illustrate specific program techniques for different types of geodata objects. The remainder of the book introduces advanced SML development techniques and script types, such as movie scripts, APPLIDATs, Macro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

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

SML in TNTmips







Be Creative with SML

SML is fully integrated into TNTmips and the other TNT products. You can create and use several types of SML scripts that provide different levels of interactivity with your geospatial data and other TNT processes. We will examine examples of all of the following script types later in this booklet:

SML Process Script: Use the SML process to create custom scripts to uniquely process your geospatial data. You can use an SML process script to apply an operation not found in the standard TNT processes or to automate a sequence of steps involving standard processes or even external programs. SML process scripts can access and create data in TNT Project Files or in external file formats. They can also create dialog windows to allow interactive setting of program options and even views of the input or output data.

Macro Script: Use a macro script to implement custom commands that operate on the geospatial data in a View window. A macro script can change the view of the data, operate on data in the View, or start an external program and provide it with data from and about the layers in the View. You can install a macro script so that it is available in any TNT View window or restrict its use to a particular saved group or layout. You can access a macro script from the View window using an icon button and from the Macros menu. Program options can be provided on a dropdown menu activated from the script's icon button and Macros menu entry.

Tool Script: Use a tool script to provide a unique interactive graphic tool to select data in a view window and apply custom processing. A tool script can provide a point, line, or polygon tool to designate the portion of the data to be processed. It can also create a dialog window to provide controls for the tool, the process, or to show the results of the processing. The tool script template provides skeletal function definitions that you can fill in to define the exact operations you need. You can activate a tool script from the View window using an icon button or from the Tools menu.

SML Layer Script: Use an SML Layer script to create a separate layer in a spatial display view with the Add SML display option. You can use such a script to render a custom cartographic element such as a magnetic declination symbol.

APPLIDAT: You can use SML to create turnkey geospatial applications for users of TNTview or TNTmips. An APPLIDAT (APPLIcation plus DATa) bundles an SML script (or scripts) with the geospatial data to be processed. Data and script are loaded together automatically when each script is run, so no file navigation is required. Launch each component script from an icon button on the APPLIDAT's toolbar, which can also include a button for instructions. An instruction set is easy to create because you can use any editor that supports the HTML format.

Run VIEWSHED.SML

The VIEWSHED.SML script is an example of an SML process script. The script and its sample data in VIEWSHED.RVC are contained on the TNT products CD-ROM and are 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 element. Many applications that deal with line-of-sight surface characteristics can use the techniques illustrated in this script.

Open the VIEWSHED.SML script in the Spatial Manipulation Language window (SML editor) 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 and classes to SML. Being aware of what functions and classes 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 interpreted SML functions and procedures, import classes written in Visual Basic or C, or invoke external programs from within SML scripts.

STEPS

☑ in the Spatial Manipulation Language window choose File / Open / *.SML File and select VIEWSHED.SML from the SML directory

☑ scroll through the script for a first look at SML

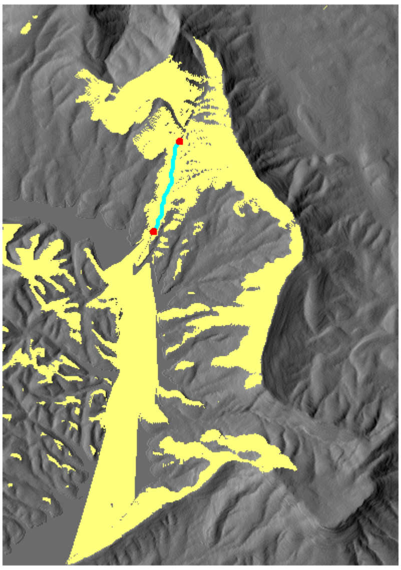
☑ click [Run...] at the bottom of the window

☑ when prompted for the input raster "RIN", select DEM from the VIEWSHED Project File in the SML directory

☑ for the input vector "V", select vpath from the VIEWSHED Project File

☑ select a new Project File and object for the output raster "ROUT"

☑ use the TNTmips Display process to view three layers: VIEWSHED / DEM, your new output raster, and VIEWSHED / VPATH



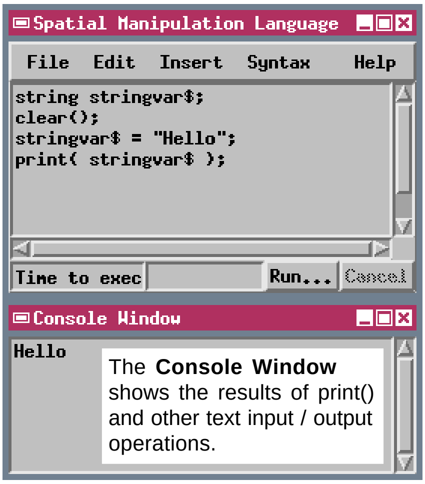
VIEWSHED.SML produces a binary raster (1's shown here in yellow) that indicates the areas visible on an elevation surface (shown here in relief shading) from a stream of points along input vector line elements (shown here in cyan). Thus if the line elements represent roads, then the yellow areas define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

Fundamentals of SML Syntax

STEPS
- ☑ clear the SML window by selecting New from the File menu
- ☑ type in the script shown in the illustration below
- ☑ click [Run] to execute the script

The **Spatial Manipulation Language** window (hereafter referred to as the "SML window") is a simple text editor that provides access to function lists and syntax checking.

An SML script can be anything from a single statement to a long structured program with nested logical branching constructs. To illustrate some of the basic elements of SML syntax, type in the sample script illustrated below. The script consists of four lines, with each line containing one program statement. The first statement declares a string variable named stringvar$. The second statement calls a predefined function that erases the contents of the Console Window. (This function requires no parameters, therefore the parentheses following the function name are empty.) The third statement assigns the string "Hello" to the previously-declared variable stringvar$. The final statement calls a predefined print function to print the value of stringvar$ to the Console Window.



```
string stringvar$;
clear();
stringvar$ = "Hello";
print( stringvar$ );
```

```
Hello
```

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

Spaces and tabs in your script are ignored when the script is interpreted by SML. Feel free to use spaces and indents to improve the clarity and readability of your scripts. For example, this script leaves spaces next to the parentheses in the print statement.

You will find the basic SML syntax rules, which are outlined on the following pages, to be fairly open and flexible. For example, most variables are created when first invoked and need not be declared before use, and use of the semicolon (;) to terminate statements is optional. Violations of the basic SML syntax rules will prevent your script from running (they are reported as errors), but you can screen such errors first by using the syntax checker. 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.

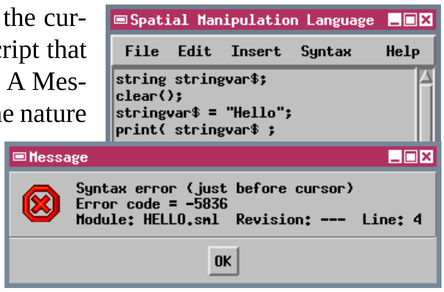
SML supports the use of various types of variables, classes, functions, and keywords, and provides standard operators for use in assignment statements and in mathematical and logical expressions.

Checking Syntax

The Check option on the Syntax menu checks your script for syntax problems. Violations of the basic SML syntax rules, such as missing function parameters, misspellings, and unclosed parentheses and loops, are reported as errors. The syntax checker cannot detect logical errors such as infinite loops or incorrect input values.

STEPS
☑ edit the previous script to remove the closing parenthesis ") " at the end of the print statement
☑ select Syntax / Check from the SML window
☑ click [OK] on the resulting Message window

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. A Message window also opens and reports the nature of the error (if possible) and the line number. Often the error immediately precedes the cursor location, but if the error involves nested processing loops, you may need to search some distance around the cursor to find the problem.



```
string stringvar$;
clear();
stringvar$ = "Hello";
print( stringvar$ ;
```

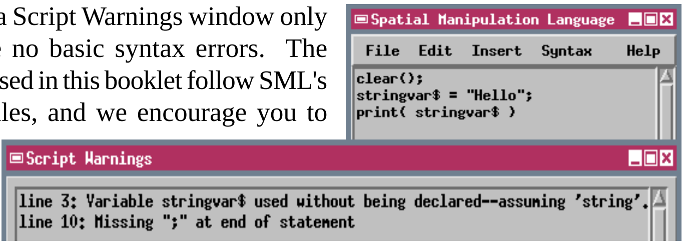
```
Syntax error (just before cursor)
Error code = -5836
Module: HELLO.sml   Revision: ---   Line: 4
```

☑ restore the closing parenthesis to the print statement
☑ delete the first statement in the script (the variable declaration) and the semicolon from the end of the last statement
☑ select Syntax / Check from the SML window
☑ click [Close] on the resulting Script Warnings window

Although the basic syntax rules are adequate for simple scripts, SML also includes an optional stricter set of syntax rules to help you ensure correct interpretation of complex, highly-structured scripts:

1. All variables must be declared before they are used in a statement.
2. Assigned variable values must match the declared variable type.
3. All statements must end in a semicolon.

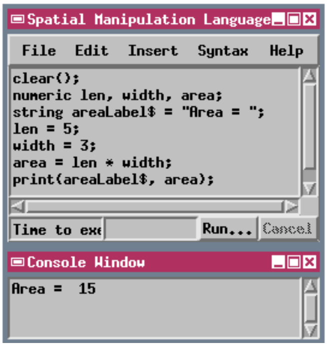
Strict syntax rules are checked only when you check syntax (not when you run the script), and violations are reported in a Script Warnings window only when there are no basic syntax errors. The sample scripts used in this booklet follow SML's strict syntax rules, and we encourage you to follow them in your own scripts.



```
clear();
stringvar$ = "Hello";
print( stringvar$ )
```

```
line 3: Variable stringvar$ used without being declared--assuming 'string'.
line 10: Missing ";" at end of statement
```

Variables



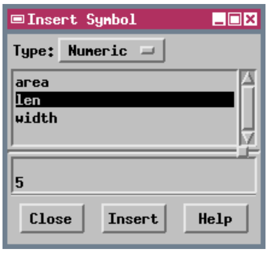
```
clear();
numeric len, width, area;
string areaLabel$ = "Area = ";
len = 5;
width = 3;
area = len * width;
print(areaLabel$, area);
```

```
Area =  15
```

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

String: initial character is lowercase; must end in '$' character if not declared. Values in assignment statements must be enclosed in double or single quotes (single quotes allow multi-line strings).

STEPS
- ☑ select File / New to clear the SML window
- ☑ type the first three lines of the script shown above and press <Enter> to start a fourth script line
- ☑ select Syntax / Check
- ☑ select Insert / Symbol and choose Numeric from the Type menu on the Insert Symbol window
- ☑ select *len* from the Numeric variable list and press [Insert]
- ☑ type in the remainder of the fourth statement
- ☑ complete the rest of the script and click [Run]



Numeric: initial character is lowercase; cannot end in '$'. Values can be integer or decimal.

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

Logical: implemented as numerics where 0 = false, and all non-zero values = true. You can use either the logical or numeric values in assignment statements. Thus

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

Array and **Class:** You must declare an array or a class before using it. Enclose an array index in square brackets:

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

You can use the Insert Symbol window (Insert / Symbol) to insert variables declared or used previously in the script. Use the Type option menu to choose a variable type (or Constant) and view the associated list. Your variable names are added to these lists when you use the Check Syntax operation or run the script. Inserting variable names rather than typing them can cut down on typing errors.

Expressions and Statements

Expressions are constructs that reduce to some value. Thus pi^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;
```

Multiple short statements can be place on a single line if separated by semicolons:

```
len = 2;  width = 5;
```

Conditional statements have the form

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

Note that the <condition> expression must be enclosed in parentheses. The *else* clause is optional, as is the "*then*":

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

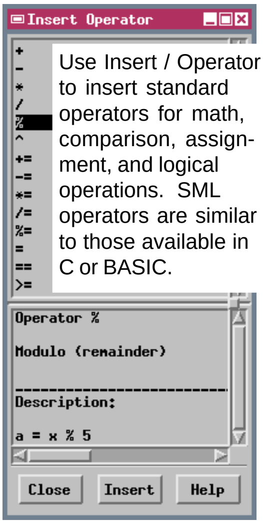
A *complex statement* involves multiple actions on separate script lines and is bracketed by the keywords "begin" and "end" in the form

```
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;
}
```

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. Use comments liberally to document the script logic for yourself and others.



Use Insert / Operator to insert standard operators for math, comparison, assignment, and logical operations. SML operators are similar to those available in C or BASIC.

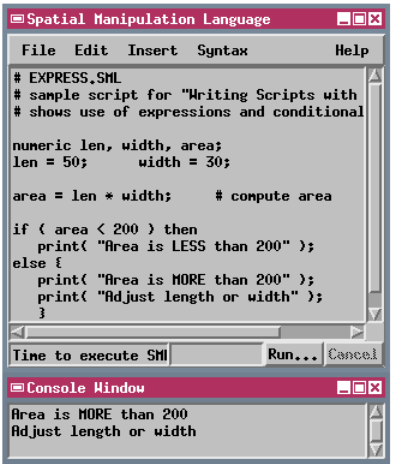
```
Operator %

Modulo (remainder)

--------------------------
Description:

a = x % 5
```

STEPS

☑ select File / Open / *.SML File and select SML / EXPRESS.SML

☑ run the script

☑ decrease the value for the width variable and run the script again



```
# EXPRESS.SML
# sample script for "Writing Scripts with
# shows use of expressions and conditional

numeric len, width, area;
len = 50;       width = 30;

area = len * width;      # compute area

if ( area < 200 ) then
   print( "Area is LESS than 200" );
else {
   print( "Area is MORE than 200" );
   print( "Adjust length or width" );
   }
```

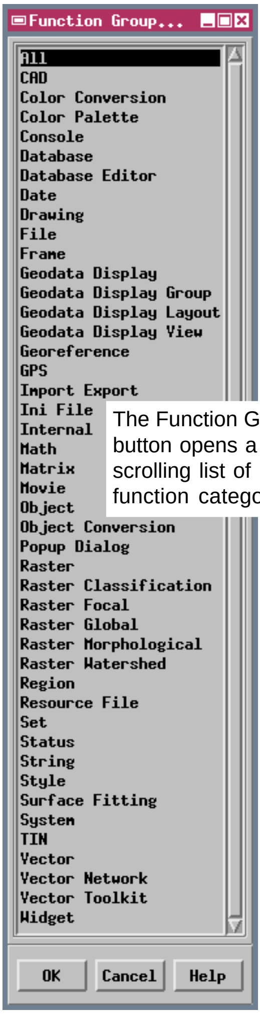
```
Area is MORE than 200
Adjust length or width
```

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

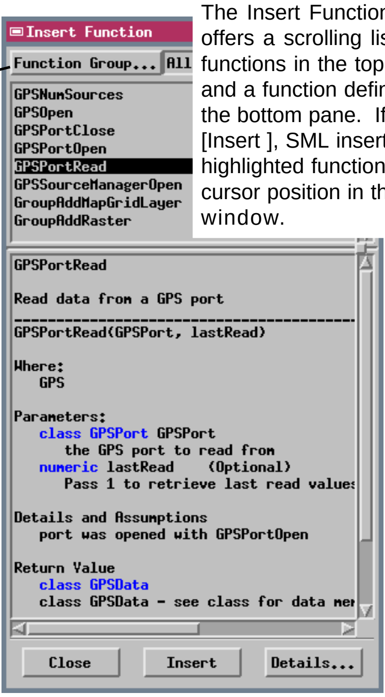
The real power of SML lies in its rich collection of built-in functions and classes that let you create, read, process, and write geospatial objects and subobjects in your TNT Project Files. Standard math functions are included along with specialized functions for display, interface, and data manipulation. 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.



```
All
CAD
Color Conversion
Color Palette
Console
Database
Database Editor
Date
Drawing
File
Frame
Geodata Display
Geodata Display Group
Geodata Display Layout
Geodata Display View
Georeference
GPS
Import Export
Ini File
Internal
Math
Matrix
Movie
Object
Object Conversion
Popup Dialog
Raster
Raster Classification
Raster Focal
Raster Global
Raster Morphological
Raster Watershed
Region
Resource File
Set
Status
String
Style
Surface Fitting
System
TIN
Vector
Vector Network
Vector Toolkit
Widget
```

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



```
GPSNumSources
GPSOpen
GPSPortClose
GPSPortOpen
GPSPortRead
GPSSourceManagerOpen
GroupAddMapGridLayer
GroupAddRaster
```

```
GPSPortRead

Read data from a GPS port
-----------------------------------------
GPSPortRead(GPSPort, lastRead)

Where:
   GPS

Parameters:
   class GPSPort GPSPort
      the GPS port to read from
   numeric lastRead     (Optional)
      Pass 1 to retrieve last read value

Details and Assumptions
   port was opened with GPSPortOpen

Return Value
   class GPSData
   class GPSData - see class for data me
```

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.

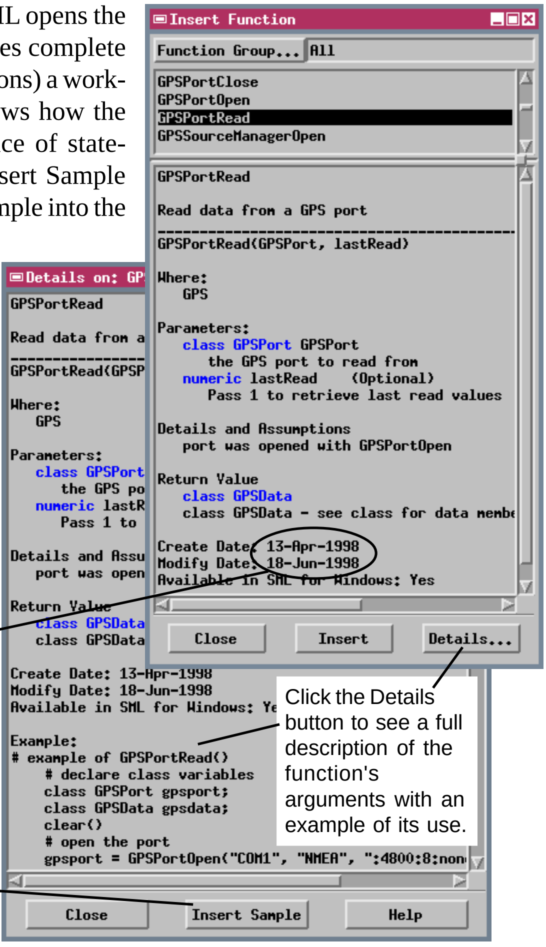
Online Function Help

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 (text in blue). You can click the Insert button to copy a complete instance of the function 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

For more information, click the Details button in the Insert Function window. SML opens the Details On: window that gives complete details, plus (for many functions) 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.



```
GPSPortRead

Read data from a GPS port
---------------------------------------------
GPSPortRead(GPSPort, lastRead)

Where:
   GPS

Parameters:
   class GPSPort GPSPort
      the GPS port to read from
   numeric lastRead     (Optional)
      Pass 1 to retrieve last read values

Details and Assumptions
   port was opened with GPSPortOpen

Return Value
   class GPSData
   class GPSData - see class for data membe

Create Date: 13-Apr-1998
Modify Date: 18-Jun-1998
Available in SML for Windows: Yes
```

```
Create Date: 13-Apr-1998
Modify Date: 18-Jun-1998
Available in SML for Windows: Ye

Example:
# example of GPSPortRead()
    # declare class variables
    class GPSPort gpsport;
    class GPSData gpsdata;
    clear()
    # open the port
    gpsport = GPSPortOpen("COM1", "NMEA", ":4800:8:non
```

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.

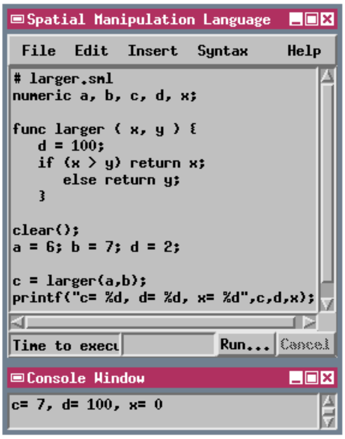
Click the Details button to see a full description of the function's arguments with an example of its use.

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

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

User-Defined Functions and Procedures

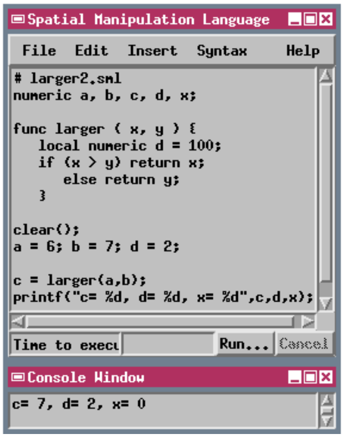
STEPS
- ☑ select File / Open / *.SML File and open LARGER.SML from the SML directory
- ☑ run the script



```
# larger.sml
numeric a, b, c, d, x;

func larger ( x, y ) {
   d = 100;
   if (x > y) return x;
      else return y;
   }

clear();
a = 6; b = 7; d = 2;

c = larger(a,b);
printf("c= %d, d= %d, x= %d",c,d,x);
```

```
c= 7, d= 100, x= 0
```

- ☑ select File / Open / *.SML File and open LARGER2.SML from the SML directory
- ☑ run the script



```
# larger2.sml
numeric a, b, c, d, x;

func larger ( x, y ) {
   local numeric d = 100;
   if (x > y) return x;
      else return y;
   }

clear();
a = 6; b = 7; d = 2;

c = larger(a,b);
printf("c= %d, d= %d, x= %d",c,d,x);
```

```
c= 7, d= 2, x= 0
```

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;  ...  }
```

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 array or class variables used in your functions and procedures must be declared before or in 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 a local variable within the function definition:

```
func  funcname  ([parmlist])  {
         local  numeric  x;  ...
```

where x is a variable name. The function parameters are exceptions to this rule; their scope is automatically limited to the function. Local variables can have the same names as global variables elsewhere in the script, though this is not recommended practice. In the examples shown at left, variable *x* retains the default value 0 in the main script because the function parameter *x* is automatically local. The assignment of value 100 to *d* in the function supercedes the value assigned before the function is called in the main script unless *d* is also defined as a local variable in the function.

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 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 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.

```
for i = 1 to NumLins(Rast) {
    for j = 1 to NumCols(Rast){
      (statement; statement; ...)
   }
}
```

While. Be careful of "while" loops.

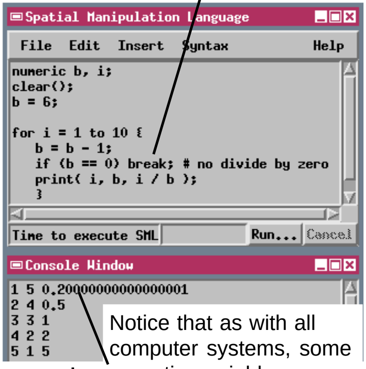
```
while (condition) statement
```

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

STEPS
- ☑ select File / Open and select WHILEFOR.SML from the SML directory
- ☑ 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 division by zero.



```
numeric b, i;
clear();
b = 6;

for i = 1 to 10 {
   b = b - 1;
   if (b == 0) break; # no divide by zero
   print( i, b, i / b );
   }
```

```
1 5 0.20000000000000001
2 4 0.5
3 3 1
4 2 2
5 1 5
```

Notice that as with all computer systems, some operations yield very small errors in floating point values (1 / 5 yields 0.20000000000000001).

Using Classes

STEPS
- ☑ clear the SML window with File / New
- ☑ select Insert / Class
- ☑ scoll through the list in the top pane of the Insert Class window and select class COLOR

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
```

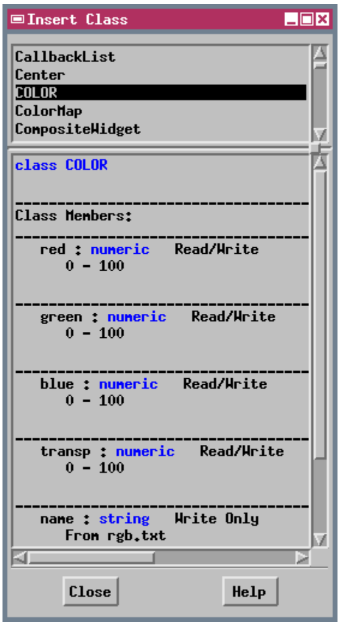
which declares background to be a class variable of the COLOR type. Members of a class are specified in the form name.member (just as database values are specified in the form table.field). For example, the class Color has five members that can be assigned values with statements in the form:

```
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

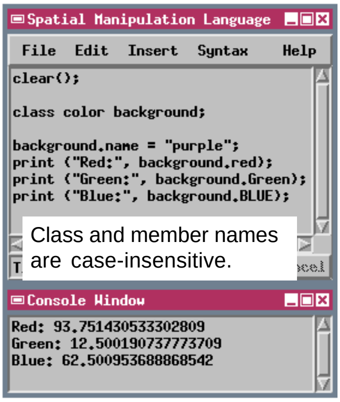
```
background.name = "purple"
```

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.



```
CallbackList
Center
COLOR
ColorMap
CompositeWidget
```

```
class COLOR

------------------------------------
Class Members:
------------------------------------
   red : numeric    Read/Write
      0 - 100

------------------------------------
   green : numeric    Read/Write
      0 - 100

------------------------------------
   blue : numeric    Read/Write
      0 - 100

------------------------------------
   transp : numeric    Read/Write
      0 - 100

------------------------------------
   name : string    Write Only
      From rgb.txt
```



```
clear();

class color background;

background.name = "purple";
print ("Red:", background.red);
print ("Green:", background.Green);
print ("Blue:", background.BLUE);
```

Class and member names are case-insensitive.

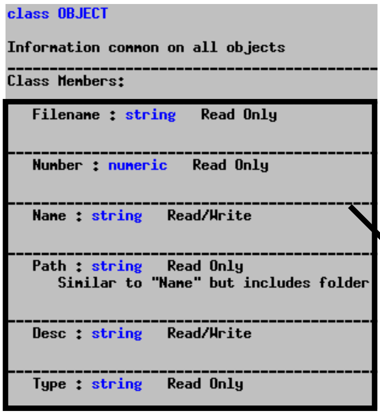
```
Red: 93.751430533302809
Green: 12.500190737773709
Blue: 62.500953688868542
```

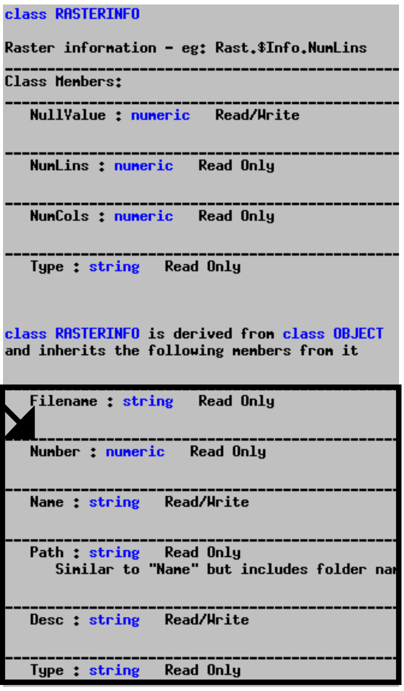
Member Inheritance and Type Checking

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 requires 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.

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


```
class OBJECT

Information common on all objects
--------------------------------------------
Class Members:

   Filename : string   Read Only
--------------------------------------------
   Number : numeric   Read Only
--------------------------------------------
   Name : string   Read/Write
--------------------------------------------
   Path : string   Read Only
      Similar to "Name" but includes folder
--------------------------------------------
   Desc : string   Read/Write
--------------------------------------------
   Type : string   Read Only
```

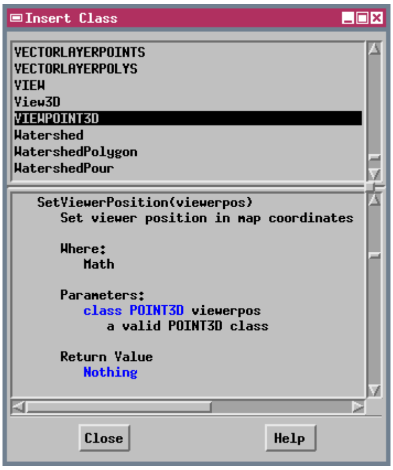

```
class RASTERINFO

Raster information - eg: Rast.$Info.NumLins
----------------------------------------------
Class Members:
----------------------------------------------
   NullValue : numeric   Read/Write
----------------------------------------------
   NumLins : numeric   Read Only
----------------------------------------------
   NumCols : numeric   Read Only
----------------------------------------------
   Type : string   Read Only

class RASTERINFO is derived from class OBJECT
and inherits the following members from it

   Filename : string   Read Only
----------------------------------------------
   Number : numeric   Read Only
----------------------------------------------
   Name : string   Read/Write
----------------------------------------------
   Path : string   Read Only
      Similar to "Name" but includes folder na
----------------------------------------------
   Desc : string   Read/Write
----------------------------------------------
   Type : string   Read Only
```

SML includes class equivalents of all of the spatial object variable types (raster, vector, CAD, TIN, and region). You can use either type of construct for the spatial objects referenced in your scripts.

Class Methods

STEPS
☑ select Insert / Class
☑ select VIEWPOINT3D in the top panel of the Insert Class window
☑ scroll the bottom panel and examine the 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.



```
VECTORLAYERPOINTS
VECTORLAYERPOLYS
VIEW
View3D
VIEWPOINT3D
Watershed
WatershedPolygon
WatershedPour
```

```
  SetViewerPosition(viewerpos)
     Set viewer position in map coordinates

     Where:
        Math

     Parameters:
        class POINT3D viewerpos
           a valid POINT3D class

     Return Value
        Nothing
```

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 ViewPos class member:

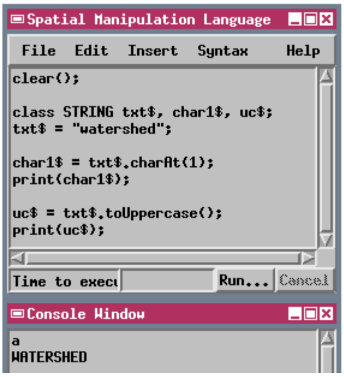
```
Class  VIEWPOINT3D  vp;
Class  POINT3D  vpos;
vpos.x  =  523487;
vpos.y  =  1473245;
vpos.z  =  2000;
vp.SetViewerPosition(vpos);
```

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

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

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$,  char1$,  uc$;
txt$  =  "watershed";

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

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



```
clear();

class STRING txt$, char1$, uc$;
txt$ = "watershed";

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

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

```
a
WATERSHED
```

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.

User Input

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.

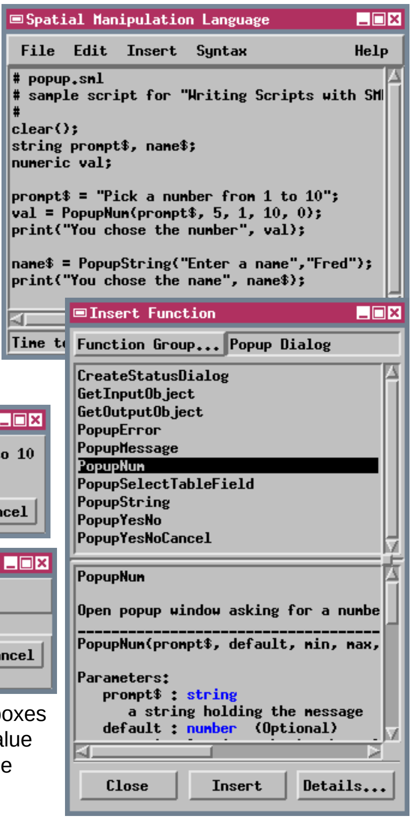
```
clear(); print("Enter your name:")
name$ = input$()
print("Your name is: ",name$)
```

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.

The Tutorial booklet *Building Dialogs in SML* provides a complete overview of procedures and techniques for creating and using your own custom dialog windows.

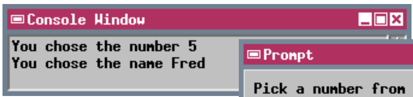
STEPS
- ☑ clear the SML window with File / New
- ☑ type in the console window prompt and input statements shown in the text and [Run] the script
- ☑ choose Insert / Function
- ☑ click the Function Group button, select Popup Dialog from the Function Group window and click [OK]
- ☑ choose File / Open and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



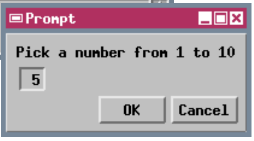
```
# popup.sml
# sample script for "Writing Scripts with SML
#
clear();
string prompt$, name$;
numeric val;

prompt$ = "Pick a number from 1 to 10";
val = PopupNum(prompt$, 5, 1, 10, 0);
print("You chose the number", val);

name$ = PopupString("Enter a name","Fred");
print("You chose the name", name$);
```

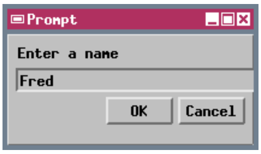
```
CreateStatusDialog
GetInputObject
GetOutputObject
PopupError
PopupMessage
PopupNum
PopupSelectTableField
PopupString
PopupYesNo
PopupYesNoCancel
```

```
PopupNum

Open popup window asking for a numbe
------------------------------------
PopupNum(prompt$, default, min, max,

Parameters:
   prompt$ : string
      a string holding the message
   default : number  (Optional)
```



```
You chose the number 5
You chose the name Fred
```





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

Using Arrays, Matrices, and Stringlists

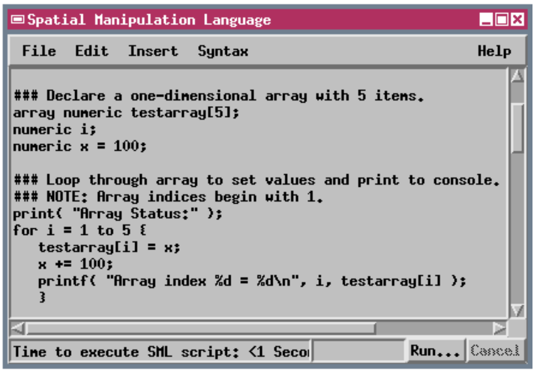
STEPS
- ☑ select File / Open and select ARRAY.SML from the SML directory
- ☑ examine the script and its comments
- ☑ click [Run] to execute the script
- ☑ examine the statements printed to the Console Window

Numeric arrays are implemented as a variable type and can be either one-dimensional or two-dimensional. When you declare an array you must specify the size of the array with a statement in the form:

```
array  numeric  arrayName[cols];
array  numeric  arrayName[rows,  cols];
```

Position within an array row or column is indicated by a subscript index number, with the first item denoted by index 1:

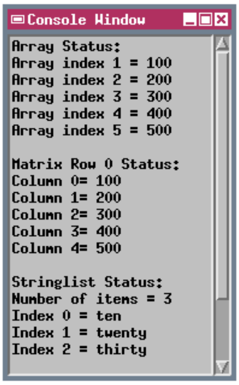
```
x  =  testArray[1]
```


Spatial Manipulation Language
File
Edit
Insert
Syntax
Help
Time to execute SML script: <1 Seco
Run...
Cancel

```
### Declare a one-dimensional array with 5 items.
array numeric testarray[5];
numeric i;
numeric x = 100;

### Loop through array to set values and print to console.
### NOTE: Array indices begin with 1.
print( "Array Status:" );
for i = 1 to 5 {
   testarray[i] = x;
   x += 100;
   printf( "Array index %d = %d\n", i, testarray[i] );
   }
```

You can resize an existing array using the functions ResizeArrayClear() (which sets all values to 0) or ResizeArrayPreserve() (which preserves existing values when the array is expanded).

Some SML vector functions that return a list of element numbers or vertex positions to an array automatically expand the array size as needed.



```
Array Status:
Array index 1 = 100
Array index 2 = 200
Array index 3 = 300
Array index 4 = 400
Array index 5 = 500

Matrix Row 0 Status:
Column 0= 100
Column 1= 200
Column 2= 300
Column 3= 400
Column 4= 500

Stringlist Status:
Number of items = 3
Index 0 = ten
Index 1 = twenty
Index 2 = thirty
```

Matrices and stringlists are implemented as classes. A matrix is always two-dimensional, so you must specify the matrix size as follows:

```
class  MATRIX  matrix;
matrix  =  CreateMatrix(rows,  cols);
```

The Matrix function group also provides functions to set and read matrix item values and to invert, transpose, and perform arithmetic operations on matrices. Matrix row-column position indices begin with 0.

A stringlist can be used to hold a list of string values. Methods in the STRINGLIST class allow you to add strings to the end or beginning of the list, to get a string by its index (beginning with 0), to remove a specified string or remove duplicates, and to sort the strings. You can also use array subscript notation to retrieve strings from the list by position.

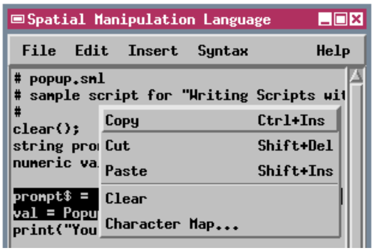
Script Development and Editing

The easiest way to develop an SML script is to adapt an existing script to the intended new task. Many sample SML scripts are distributed with the TNT products in the SCRIPTS directory, which has subdirectories for different categories of scripts (vector, database, tool scripts, and others). You can also use any of the examples from this tutorial booklet as starting points for your own scripts.

You can open two SML script editing windows side by side and use the SML copy and paste functions to copy sections of code from an existing script into the script you are developing. You can access the SML cut, copy, and paste functions from the Edit menu on the SML window or from a pop-up menu that opens when you press the right mouse button (with the cursor within the editing pane). If you are running under the Windows or Mac OS X operating system, these SML functions use the operating system's clipboard, so you can also cut and paste text between the SML editor and another text editor.

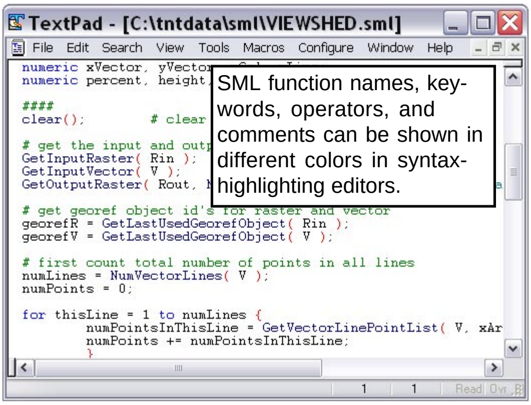
A number of low-cost or freeware text editing programs provide color-coded syntax highlighting for various programming languages. The MicroImages website (www.microimages.com/gvim) provides links to a number of these editors as well as configuration files to enable SML syntax highlighting. The available editors include UltraEdit and TextPad (Windows), Vim (various platforms) and Hydra (Mac OS X 10.2 and higher). Although these editors can highlight syntax, they do not provide access to the SML syntax checker.

STEPS
- ☑ 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 it does not obscure the first one
- ☑ select several lines of code from the first script
- ☑ use the Copy and Paste options on the Edit menus to copy the selected section to the new script
- ☑ choose File / Exit for the new script window and do not save changes



```
# popup.sml
# sample script for "Writing Scripts wit
#
clear();
string pro
numeric va

prompt$ =
val = Popu
print("You
```

Right mouse-button menu with Copy, Cut, and Paste options


TextPad - [C:\tntdata\sml\VIEWSHED.sml]
File Edit Search View Tools Macros Configure Window Help
1 1 Read Ovr

```
numeric xVector, yVecto
numeric percent, height,

####
clear();          # clear

# get the input and outp
GetInputRaster( Rin );
GetInputVector( V );
GetOutputRaster( Rout, N

# get georef object id's for raster and vector
georefR = GetLastUsedGeorefObject( Rin );
georefV = GetLastUsedGeorefObject( V );

# first count total number of points in all lines
numLines = NumVectorLines( V );
numPoints = 0;

for thisLine = 1 to numLines {
        numPointsInThisLine = GetVectorLinePointList( V, xAr
        numPoints += numPointsInThisLine;
        }
```

SML function names, keywords, operators, and comments can be shown in different colors in syntax-highlighting editors.

Preprocessor Commands and Debugging

STEPS
☑ select File / Open and select DEBUG.SML from the SML directory
☑ select Insert Keyword
☑ scroll to the bottom of the list in the Insert Keyword window to see the SML preprocessor directives
☑ close the Insert Keyword window
☑ scroll through the script to see how the preprocessor commands are used
☑ [Run] the script, following the steps on page 5
☑ examine the values printed to the Console Window

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

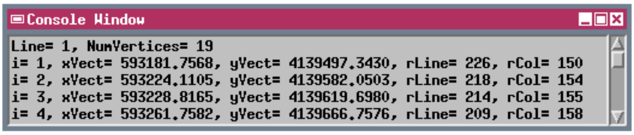
While you are developing a complex script you might want to have a "normal" mode of execution and a "debug" mode. In debug mode the current values of variables would be printed to the console at various points in the script to help you verify correct execution of intermediate steps and/or identify points 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, deactivating all of your debugging code but leaving it in place for later use. The script in this exercise is a version of the VIEWSHED script that illustrates the use of printf() statements in a debug mode.



```
Line= 1, NumVertices= 19
i= 1, xVect= 593181.7568, yVect= 4139497.3430, rLine= 226, rCol= 150
i= 2, xVect= 593224.1105, yVect= 4139582.0503, rLine= 218, rCol= 154
i= 3, xVect= 593228.8165, yVect= 4139619.6980, rLine= 214, rCol= 155
i= 4, xVect= 593261.7582, yVect= 4139666.7576, rLine= 209, rCol= 158
```

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

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

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

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

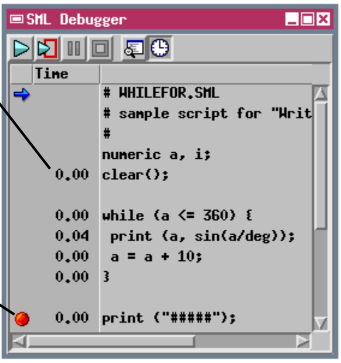
The included script should be in the same directory or Project File as the parent script. If you have several scripts that need to use the same user-defined function, the function definition can be in a separate script that you "$include" in the other scripts.

SML Debugger and Script Timing

The SML Debugger window provides a specialized script execution environment designed to help you analyze and debug a complex script. Icon buttons on the window let you run the script as usual or step through it one statement at a time. As the script executes, a blue arrow symbol moves down in the left column of the window to show the current execution step. You can also insert temporary break points by clicking in the left column of the window. Execution of the script stops automatically whenever a break point is encountered. You can restart execution after the break using the Run or Step icon buttons. You can remove a breakpoint by clicking on its symbol.

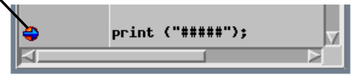
The SML Debugger window can also show the execution time (in seconds to hundredth-second accuracy) for each script step. For user-defined functions and procedures, cumulative times for one or more function calls are shown with the function definition, not where it is called in the script. You can use this tool to determine whether you can improve the speed and efficiency of your script.

Execution times are shown in the expanded left column. Times less than .005 second are shown as 0.00.



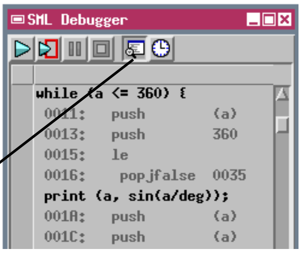
```
      # WHILEFOR.SML
      # sample script for "Writ
      #
      numeric a, i;
0.00  clear();

0.00  while (a <= 360) {
0.04   print (a, sin(a/deg));
0.00   a = a + 10;
0.00  }

0.00  print ("#####");
```

Click in the left column to place a temporary break point where script execution will automatically stop.



```
print ("#####");
```

Turn on the Show Pseudo Code icon button to expand the script view to show pseudo assembly code generated for each script statement.

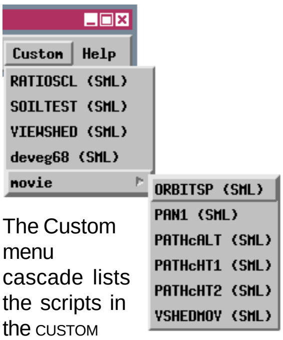


```
while (a <= 360) {
 0011:   push        (a)
 0013:   push        360
 0015:   le
 0016:    popjfalse  0035
print (a, sin(a/deg));
 001A:   push        (a)
 001C:   push        (a)
```

STEPS

- ☑ select File / Open and choose WHILEFOR.SML from the SML directory
- ☑ select File / Debug
- ☑ in the SML Debugger window, press the Show Pseudo Code icon button, examine the code, then press again to restore the normal script view
- ☑ scroll down to the `print("#####")` statement and left-click in the left column (yellow) to place a break point (red symbol) next to it
- ☑ press the Run icon button in the Debugger window; note the blue arrow indicator stops at the break point
- ☑ click on the break point to clear it
- ☑ press the Show Timing icon button
- ☑ press the Step icon button ten times; note the repeat of the "for" loop
- ☑ press the Stop icon button
- ☑ close the SML Debugger window using the X icon button in the window title bar

Toolbars and the SML Custom Menu

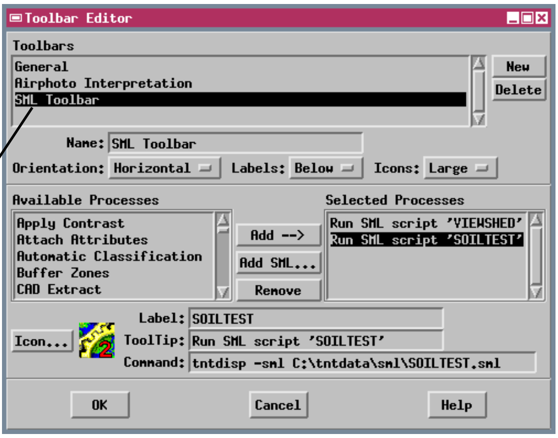


The Custom menu cascade lists the scripts in the CUSTOM directory in your TNT installation directory.

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 add frequently-used SML scripts to the TNT main menu. Simply create a directory named *custom* in your main TNT directory. Each script you place in this directory then appears as an entry on a Custom menu on the TNT main menu. Scripts in subdirectories in the *custom* directory appear on submenus on the Custom menu. Selecting a script from the menu runs the script.

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.

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 SML / VIEWSHED.SML
- ☑ click [Icon...] and select an icon
- ☑ repeat the previous two steps for SML / SOILTEST.SML
- ☑ click [OK] to finish


Toolbar Editor

Toolbars
General
Airphoto Interpretation
SML Toolbar

New
Delete

Name: SML Toolbar
Orientation: Horizontal Labels: Below Icons: Large

Available Processes
Apply Contrast
Attach Attributes
Automatic Classification
Buffer Zones
CAD Extract

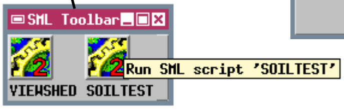
Add -->
Add SML...
Remove

Selected Processes
Run SML script 'VIEWSHED'
Run SML script 'SOILTEST'

Icon...
Label: SOILTEST
ToolTip: Run SML script 'SOILTEST'
Command: tntdisp -sml C:\tntdata\sml\SOILTEST.sml

OK Cancel Help

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



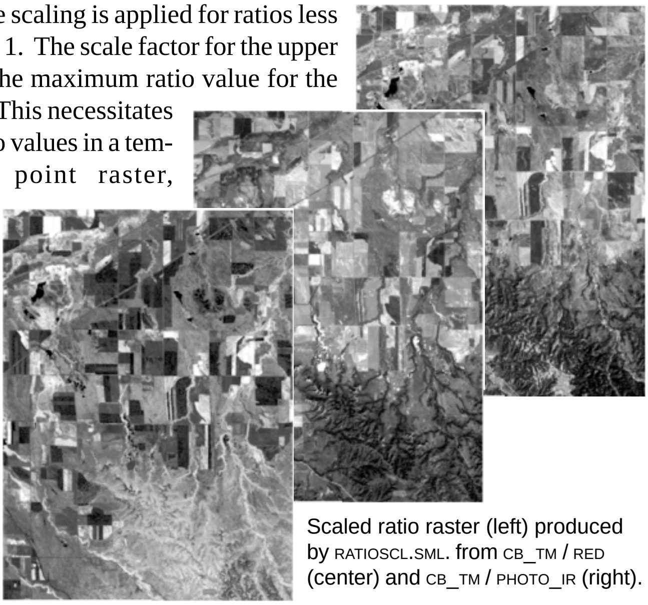
Raster Objects

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 SML directory
- ☑ 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 CB_DATA
- ☑ select RED from the CB_TM Project File for input object D
- ☑ create a new raster object for RATIOSCL
- ☑ for this exercise and those on the following pages, use the Display process to display the input object(s) and the new object(s) created by the script

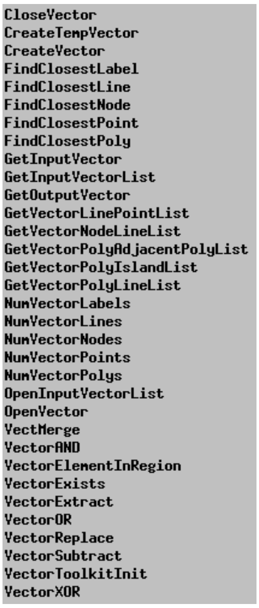

Scaled ratio raster (left) produced by RATIOSCL.SML. from CB_TM / RED (center) and CB_TM / PHOTO_IR (right).

Vector Objects

STEPS

☑ select File / Open / *.SML File and open the script VECTCOMB.SML from the SML directory

☑ 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.

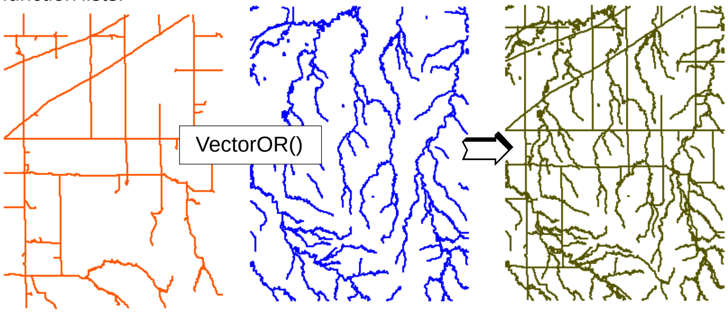
A growing list of functions support vector object creation, reading, writing, and manipulation. Look for vector function definitions in the Vector, Vector Network, and Vector Toolkit groups.

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 SCRIPTS / VECTOR / VECEXTR.SML from the TNT Products CD.

SML also supports more complex interaction between vector objects and objects of other types. You have already seen VIEWSHED.SML (page 5). Another example is provided in SCRIPTS / 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.



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

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:

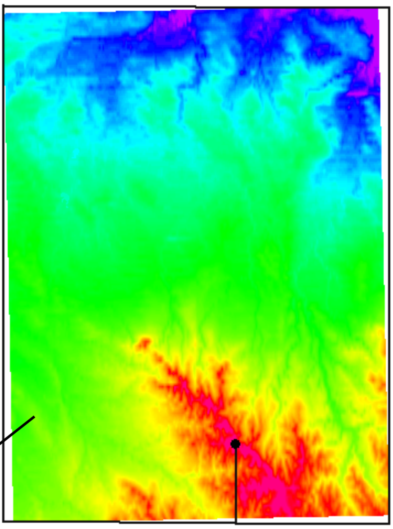
```
GetInputVector(V);
VectorToolkitInit(V);
     [Editing operations with vector
             toolkit functions]
CloseVector(V);
```

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.

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

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


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

CAD and TIN Objects

STEPS

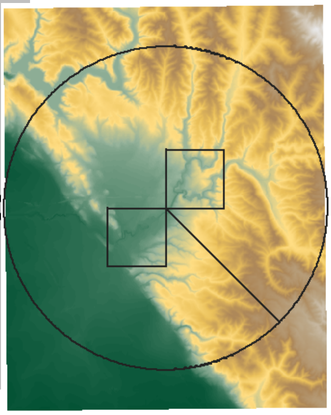
- ☑ select File / Open / *.SML File and open the script SML / CAD.SML
- ☑ examine and then run the script using raster object HAYWARD from the HAYWDEM Project File in SF_DATA
- ☑ open the script SML / TIN.SML
- ☑ study and then run the script, using object ELEV_PTS from the SURFACE Project File in the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

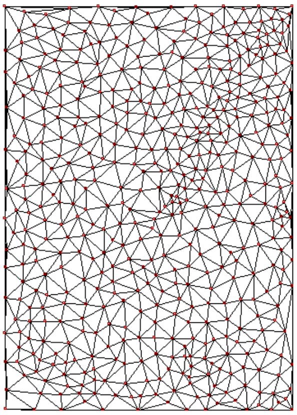
A growing list of functions support CAD and TIN object creation, reading, writing, and manipulation. Sample script CAD.SML uses some of the numerous CAD functions. The script uses a raster object as input to define geographic extents and georeferencing and creates a new georeferenced CAD object to which 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
```

```
CADAttachDBRecord
CADCreateBlock
CADElementInRegion
CADElementType
CADGetElementList
CADInsertBlock
CADNumBlocks
CADNumElements
CADReadArc
CADReadArcChord
CADReadArcWedge
CADReadBox
CADReadCircle
CADReadEllipse
CADReadEllipticalArc
CADReadEllipticalArcChord
CADReadEllipticalArcWedge
CADReadLine
CADReadPoint
CADReadPoly
CADReadText
CADUnattachDBRecord
CADWriteArc
CADWriteArcChord
CADWriteArcWedge
CADWriteBox
CADWriteCircle
CADWriteEllipse
CADWriteEllipticalArc
CADWriteEllipticalArcCh
CADWriteEllipticalArcWe
CADWriteLine
CADWritePoint
CADWritePoly
CADWriteText
CloseCAD
CreateCAD
GetInputCAD
GetOutputCAD
OpenCAD
```





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

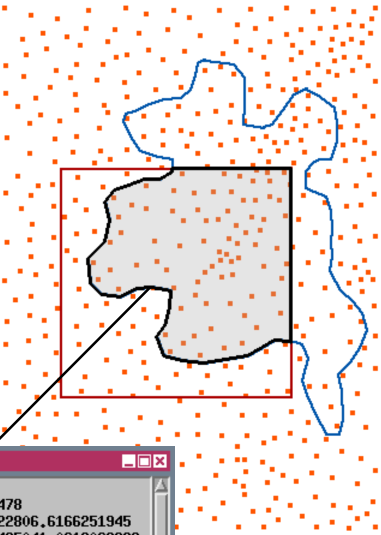
```
for each RastVar in RegionVar {
   [actions]
}
```

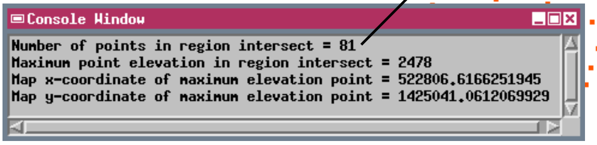
restricts the actions to raster cells that lie within the region boundaries. This construction provides a simpler alternative to using values in a binary mask raster to control the operations.

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 the SML directory
- ☑ study the script structure and comments
- ☑ run the script using for input the region objects POLYREGION and RECTANGLE from the SML / REGION Project File and vector object ELEV_PTS from the SURFMODL / SURFACE Project File

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





```
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

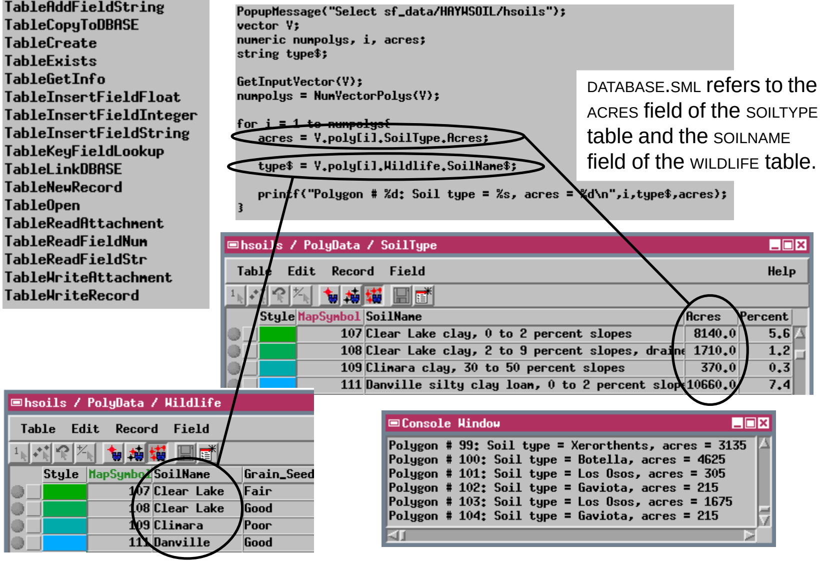
- ☑ open the sample script DATABASE.SML from the SML directory
- ☑ run the script using object HSOILS from the HAYWSOIL Project File in the SF_DATA directory for input
- ☑ open the sample script DB2.SML from the SML directory
- ☑ run the script using object CB_SOILSLITE from the CB_SOILS Project File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

```
DatabaseGetTableInfo
FieldGetInfoByName
FieldGetInfoByNumber
NumRecords
OpenCADDatabase
OpenDatabase
OpenRasterDatabase
OpenTINDatabase
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.



```
PopupMessage("Select sf_data/HAYWSOIL/hsoils");
vector V;
numeric numpolys, i, acres;
string type$;

GetInputVector(V);
numpolys = NumVectorPolys(V);

for i = 1 to numpolys{
   acres = V.poly[i].SoilType.Acres;

   type$ = V.poly[i].Wildlife.SoilName$;

   printf("Polygon # %d: Soil type = %s, acres = %d\n",i,type$,acres);
}
```

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

hsoils / PolyData / SoilType

Style	MapSymbol	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, draine	1710.0	1.2
🟩	109	Climara clay, 30 to 50 percent slopes	370.0	0.3
🟦	111	Danville silty clay loam, 0 to 2 percent slop	10660.0	7.4

hsoils / PolyData / Wildlife

Style	MapSymbol	SoilName	Grain_Seed
🟩	107	Clear Lake	Fair
🟩	108	Clear Lake	Good
🟩	109	Climara	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.

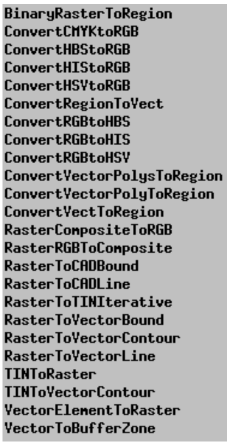
The SOILTEST.SML sample script automates the processing of soil sample data and uses several types of object conversion functions. The script reads a series of soil chemistry values stored in a database table attached to input vector point elements representing sample locations. For each type of value (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.

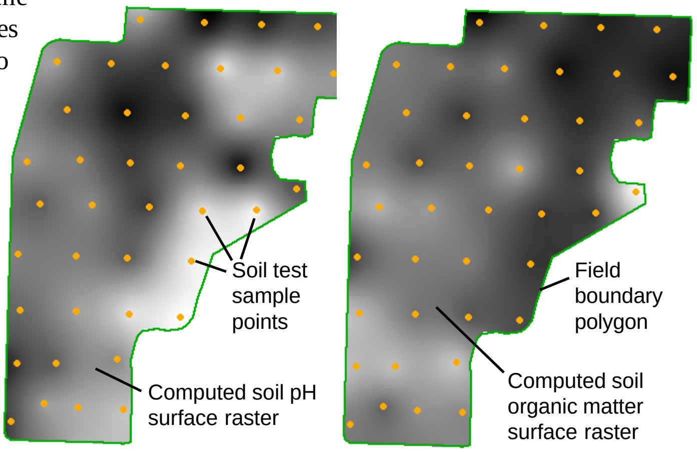
STEPS

☑ open the sample script SOILTEST.SML from the SML directory

☑ study the script, then run it using objects in the SML / SOILTEST Project File 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

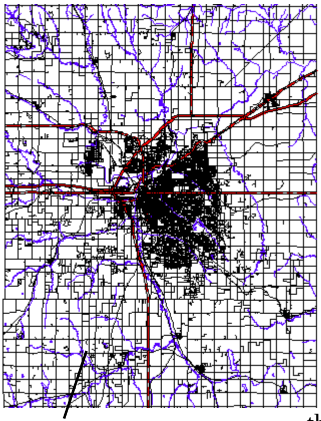

```
BinaryRasterToRegion
ConvertCMYKtoRGB
ConvertHBStoRGB
ConvertHIStoRGB
ConvertHSVtoRGB
ConvertRegionToVect
ConvertRGBtoHBS
ConvertRGBtoHIS
ConvertRGBtoHSV
ConvertVectorPolysToRegion
ConvertVectorPolyToRegion
ConvertVectToRegion
RasterCompositeToRGB
RasterRGBToComposite
RasterToCADBound
RasterToCADLine
RasterToTINIterative
RasterToVectorBound
RasterToVectorContour
RasterToVectorLine
TINToRaster
TINToVectorContour
VectorElementToRaster
VectorToBufferZone
```


Soil test sample points
Computed soil pH surface raster
Field boundary polygon
Computed soil organic matter surface raster

Sample Script: Extract Polygons

STEPS
- ☑ choose File / Open / *.SML File and select SML / TIGER1.SML
- ☑ study the script structure and comments

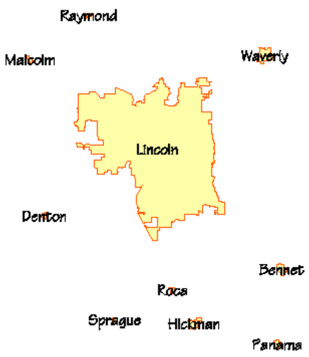
The sample script TIGER1.SML provides an example of vector and database processing in SML. It extracts specified lines from an input vector object, writes them into an output vector object, and transfers input line attributes to output polygon attributes.



TIGER vector for a single county with lines styled based on their attributes. TIGER files are available for free download at **www.census.gov**.

TIGER1.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.

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 an input TIGER vector object 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. A multi-input version of 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.

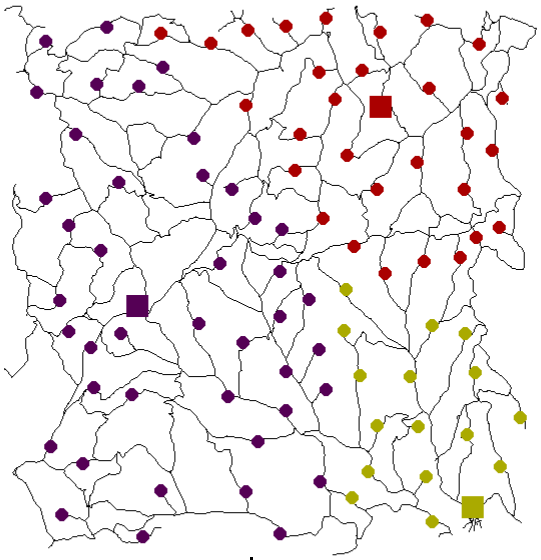
More about the extract polygon script is available in an online document at
http://www.microimages.com/relnotes/v65/smltiger.pdf

Sample Script: Network Routing

The sample script NETWORK1.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 SML / NETWORK1.SML
- ☑ study the script structure and comments
- ☑ run the script using objects FARMS, PLANTS, and ROADS from the SML / NETWORK1 Project File



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

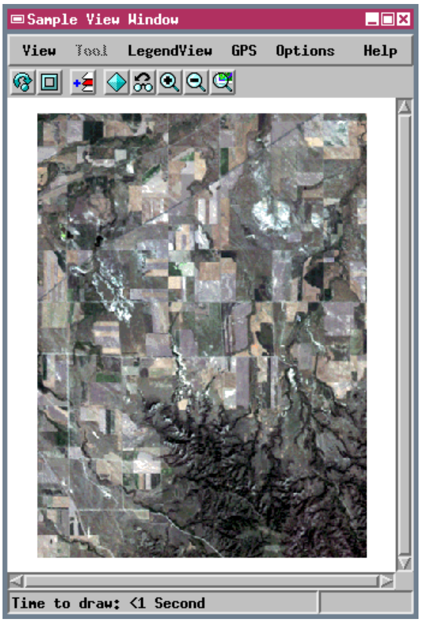
http://www.microimages.com/relnotes/v65/smlnz.pdf

Creating and Opening a View Window

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

An SML script can create and open a View window to display input or output objects used in the script. The View can also be used to provide user interaction with the objects via the standard graphical tools found in the Display process.

View windows are created using the Motif widget (dialog component) set that is used to create all of the windows in the X Windows versions of the TNT products. In Motif, any window (dialog) begins as an instance of class XmForm, a generic container widget. 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.



Time to draw: <1 Second

Sample script VIEW.SML shows the basic steps required to open a view window of a group and display an input raster. The script in the next exercise displays several data layers and provides a graphical point tool for obtaining coordinate information from the View. The movie scripts and APPLIDAT scripts discussed subsequently provide further examples.

SML also provides another, simpler way to provide user interaction between a script and data in a View. Tool Scripts and Macro Scripts can be launched from a View window in the Spatial Data Display process and can automatically access and operate on the objects in the View. These scripts are discussed in detail in a later section of this booklet.

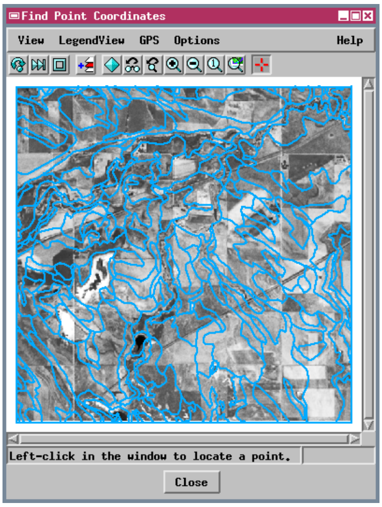
For more information about creating dialog windows consult the tutorial booklet *Building Dialogs in SML*.

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 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 obejcts 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.



Left-click in the window to locate a point.

```
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
```

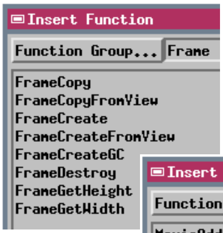
Movie Generation Scripts

STEPS

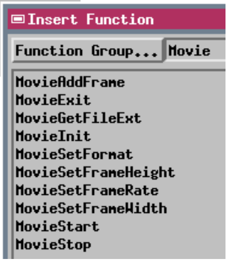
- ☑ choose File / Open / *.SML File and select SCRIPTS / MOVIE / VSHEDMOV.SML
- ☑ study the script structure and comments

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.

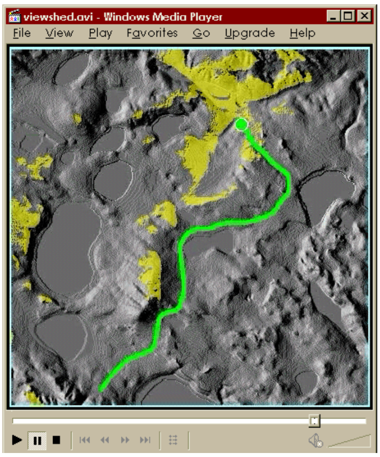


```
FrameCopy
FrameCopyFromView
FrameCreate
FrameCreateFromView
FrameCreateGC
FrameDestroy
FrameGetHeight
FrameGetWidth
```



```
MovieAddFrame
MovieExit
MovieGetFileExt
MovieInit
MovieSetFormat
MovieSetFrameHeight
MovieSetFrameRate
MovieSetFrameWidth
MovieStart
MovieStop
```

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 the view window 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/v65/moviesml.pdf

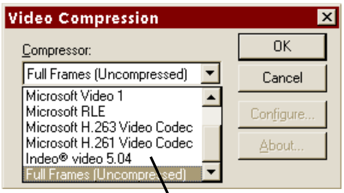
3D Simulation Scripts

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.

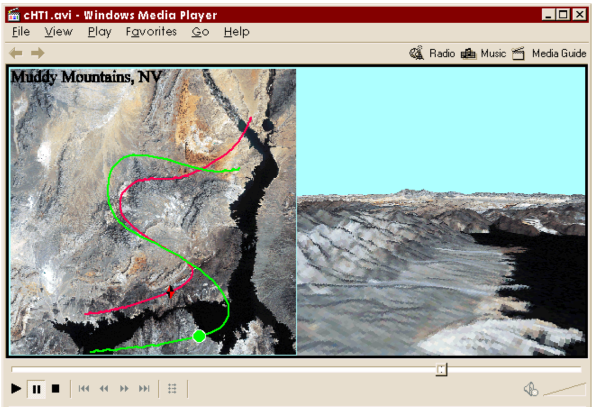
Class members and methods in the VIEWPOINT3D class are used to manipulate the settings for the 3D view. Each 3D view has a viewer position and a position that the viewer is looking at, the point where the current view is centered. SML gives you complete control over both positions. You can set viewer 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 SCRIPTS / MOVIE / PATHCHT1.SML
- ☑ study the script structure and comments


Video Compression
Compressor:
Full Frames (Uncompressed)
Microsoft Video 1
Microsoft RLE
Microsoft H.263 Video Codec
Microsoft H.261 Video Codec
Indeo® video 5.04
Full Frames (Uncompressed)
OK
Cancel
Configure...
About...

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.


cHT1.avi - Windows Media Player
File View Play Favorites Go Help
Radio Music Media Guide
Muddy Mountains, NV

Movies created from these sample SML movie scripts can be downloaded from

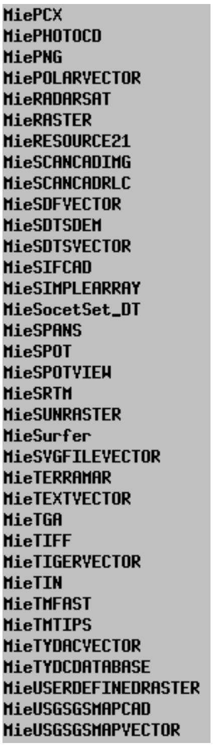
http://www.microimages.com/promo/smlmovies

Batch Import with SML

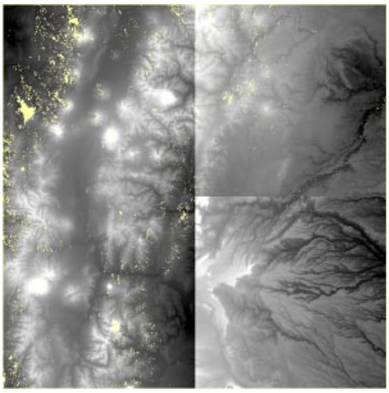
STEPS
- ☑ choose File / Open / *.SML File and select SCRIPTS / RASTER / IMPORT_SRTM.SML
- ☑ study the script structure and comments
- ☑ choose Insert / Class
- ☑ scroll down in the list in the Insert Classes window to class MieUSERDEFINEDRASTER
- ☑ examine some of the other Mie classes
- ☑ close the Insert Class window

You can use an SML script to automate repetitive tasks such as importing tens or hundreds of data files with the same format. TNTmips supports the import or export of dozens of external file formats. The program code needed to import or export each of these formats is encapsulated in SML as a class structure beginning with the letters "Mie". For example, class MieGeoTIFF supports the import or export of GeoTIFF images. The Mie classes are used in conjunction with functions in the Import Export function group that allow you to import to or export from a specific object type (raster, vector, CAD, database, or TIN). Class members for each Mie class (or a parent class) allow you to set process parameters such as raster size, compression, vector topology type, and others.

The script for this exercise performs a batch import of raw, ungeoreferenced surface height files produced by NASA's Shuttle Radar Topography Mission (www.jpl.nasa.gov/srtm). The script uses the MieUSERDEFINEDRASTER class to import the elevation grids to rasters. Each height file is one degree of latitude and longitude in extent. The script reads the latitude and longitude of the southwest tile corner from the file name and georeferences each imported raster by creating control points at the corners of the raster.

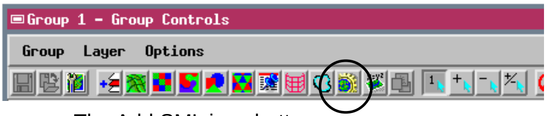

MiePCX
MiePHOTOCD
MiePNG
MiePOLARVECTOR
MieRADARSAT
MieRASTER
MieRESOURCE21
MieSCANCADIMG
MieSCANCADRLC
MieSDFVECTOR
MieSDTSDEM
MieSDTSVECTOR
MieSIFCAD
MieSIMPLEARRAY
MieSocetSet_DT
MieSPANS
MieSPOT
MieSPOTVIEW
MieSRTM
MieSUNRASTER
MieSurfer
MieSVGFILEVECTOR
MieTERRAMAR
MieTEXTVECTOR
MieTGA
MieTIFF
MieTIGERVECTOR
MieTIN
MieTMFAST
MieTMTIPS
MieTYDACVECTOR
MieTYDCDATABASE
MieUSERDEFINEDRASTER
MieUSGSGSMAPCAD
MieUSGSGSMAPVECTOR

Some of the many SML classes available to automate Import / Export tasks.



Four one-degree SRTM height tiles (covering part of Ecuador) imported with the script in this exercise. The raw height grids include numerous null (no data) cells (yellow in this image) on slopes facing away from the radar sensor.

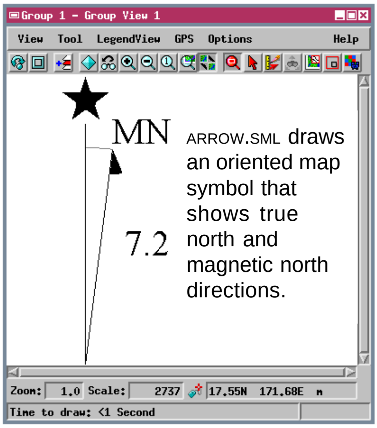
SML Layer in Display



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.

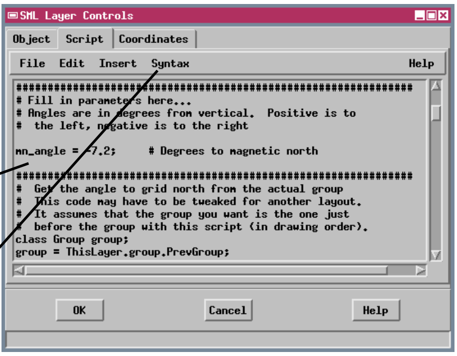
The sample script ARROW.SML is designed to draw an oriented magnetic declination map symbol in a layout. The SML layer should be alone in a group. It determines the true north direction from the previous map group in the layout. Sample script neatline.sml draws a neatline around a group, and includes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



ARROW.SML draws an oriented map symbol that shows true north and magnetic north directions.

STEPS

- ☑ run Display / Spatial Data and open a new 2D Group
- ☑ click Add SML in the Group Controls window
- ☑ select the Script tab in the SML Layer Controls window and choose File / Open / *.SML
- ☑ select 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 _8BIT from the CB_COMP Project File in CB_DATA
- ☑ click Add SML and select 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



```
###############################################################
# Fill in parameters here...
# Angles are in degrees from vertical.  Positive is to
#  the left, negative is to the right

mn_angle = -7.2;     # Degrees to magnetic north

###############################################################
#  Get the angle to grid north from the actual group
#  This code may have to be tweaked for another layout.
#  It assumes that the group you want is the one just
#  before the group with this script (in drawing order).
class Group group;
group = ThisLayer.group.PrevGroup;
```

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.

SML and GeoFormulas

See the tutorial booklet *Using Geospatial Formulas* for a complete introduction to constructing and using GeoFormulas

STEPS
- ☑ remove the SML layer from the display group
- ☑ click Add Geoformula / Quick-Add Geoformula
- ☑ select 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 CB_DATA

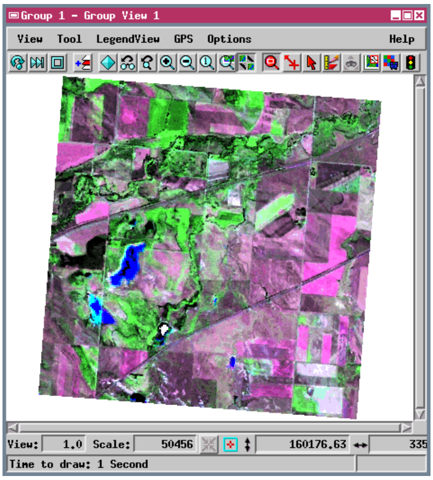
GEOFRMLA / BROV_UMN.GSF illustrates the dynamic enhancement of low-resolution TM imagery with a high-resolution SPOT image.

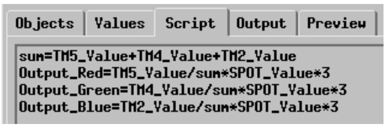
A GeoFormula layer is a computed display layer that uses one or more input objects to derive a result for display. It gives you a way to apply SML manipulations to objects "on the fly" rather than running separate processes to prepare output objects for display. A GeoFormula layer contains a "virtual 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.





```
sum=TM5_Value+TM4_Value+TM2_Value
Output_Red=TM5_Value/sum*SPOT_Value*3
Output_Green=TM4_Value/sum*SPOT_Value*3
Output_Blue=TM2_Value/sum*SPOT_Value*3
```

☑ close the display group when you have completed this exercise

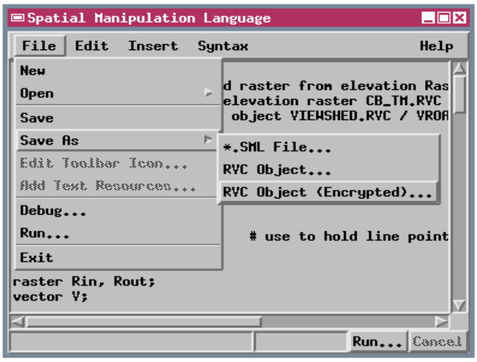
Script Objects and Encryption

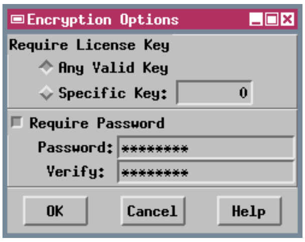
So far you have worked with SML scripts that have been saved as independent text files with the SML file extension. These are 1-byte text files that can be opened with any text editor. If you do edit a script file with another editor, be sure to save it with the SML extension.

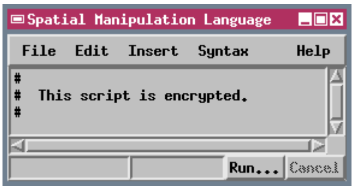
An SML script also can be saved as a script object in a Project File (use File / Save As / RVC Object). This allows you to put input, output, and script objects all in the same file if you find this more convenient. Another advantage to storing a script in a Project File is the ability to **encrypt** 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.

STEPS
- ☑ select File / Open / *.SML File and choose SML / VIEWSHED.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 (the SML window then 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.


```
d raster from elevation Ras
elevation raster CB_TM.RVC
 object VIEWSHED.RVC / VROA

 # use to hold line point
raster Rin, Rout;
vector V;
```


Require License Key
◇ Any Valid Key
◇ Specific Key: 0
☐ Require Password
Password: ********
Verify: ********

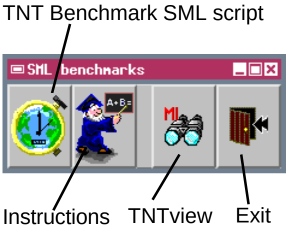

```
#
#  This script is encrypted.
#
```

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**.

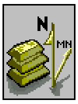
APPLIDATs

STEPS
- ☑ choose Process / SML / Run
- ☑ select SCRIPTS / 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



- ☑ select File / Open / RVC Object in the SML window
- ☑ select SML / SMLLAYER.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 called 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. 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.

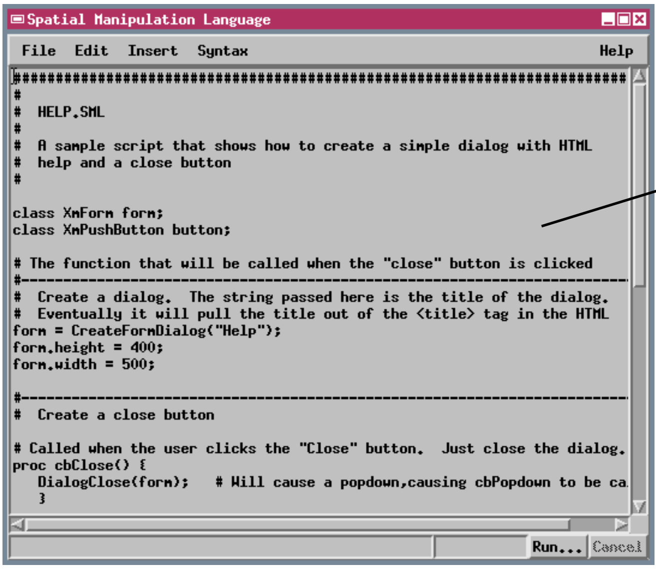
Providing APPLIDAT Instructions

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.

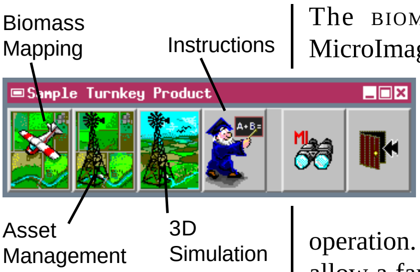
STEPS
- ☑ choose File / Open / *.SML File in the SML window
- ☑ select BENCHMRK.SML in the Select File window
- ☑ select the Help script object in the Select Object window
- ☑ examine the script structure and comments



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

class XmForm form;
class XmPushButton 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



STEPS

- ☑ select Support / Maintenance / Project File from the TNTmips main menu and examine the contents of BIOMASS2.SML in the SCRIPTS / APPLIDAT directory
- ☑ exit from Project File Maintenance and select Process / SML / Run
- ☑ choose 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

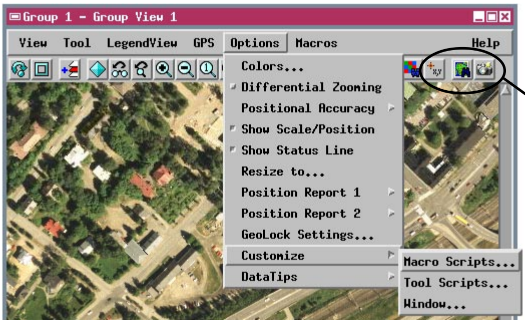
The BIOMASS2 APPLIDAT was written by MicroImages to provide an example and prototype of a turnkey APPLIDAT product. It illustrates how an APPLIDAT can let the user carry out a series of operations on the input data and automatically pass intermediate products along to the next operation. In this example 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 SML scripts launched from a View window that 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. You can set up a general-purpose tool script or macro script to be available from any type of 2D View window or save a data-specific script with a particular group or layout. Scripts saved with a layout that becomes part of an atlas are also available for use in TNTatlas. 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 can be launched from icon buttons on a View window's toolbar or from the Tool and Macros menus.

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 processsing itself.

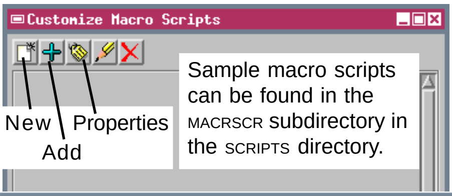
Macro scripts and tool scripts:

- are executed from an icon button on a View window toolbar or from a menu;
- 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.

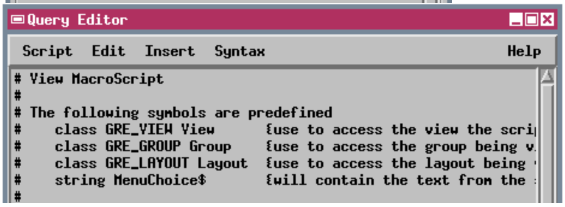
A tool script invokes a drawing tool and/or a dialog window (defined by the script-writer) that allow the user to interact with the spatial data in the view window. For example, the user could outline an area or select particular elements to be processed. A macro script does not allow such graphical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

Macro Script Setup

Macro scripts can be launched from the View window's Macros menu (which appears once you have installed a macro script) or from an optional icon button on the toolbar. To add a macro script, choose Options / Customize / Macro Scripts from the View window, which 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.



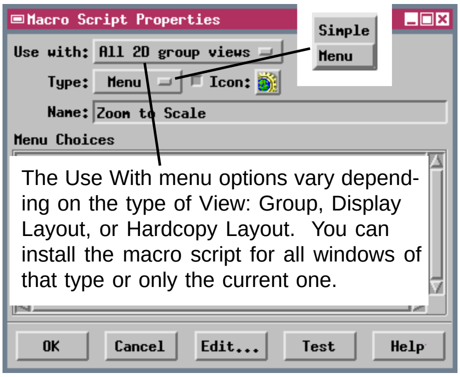
Sample macro scripts can be found in the MACRSCR subdirectory in the SCRIPTS directory.



```
# View MacroScript
#
# The following symbols are predefined
#    class GRE_VIEW View        {use to access the view the scri
#    class GRE_GROUP Group      {use to access the group being v
#    class GRE_LAYOUT Layout    {use to access the layout being
#    string MenuChoice$         {will contain the text from the
#
```

Once you have created or added the macro script, the Macro Script Properties window opens. You can choose whether the script is accessible from all Views of the current type or only from the current saved group or layout. Choose Simple from the Type menu to have your tool script execute automatically without further input from the user. Choose Menu if you want drop-down choices presented from the Macros menu entry and the icon button; the Menu Choices text field then becomes active so you can enter the menu choices needed for the script. Use the Icon toggle button to indicate whether a script icon button appears on the View window toolbar; click on the default icon to open a dialog to select an appropriate icon. The text you enter in the Name field is used for the menu entry in the Macros menu and for the script icon button's ToolTip.

The Test button at the bottom of the window lets you run your script without closing the setup windows.



The Use With menu options vary depending on the type of View: Group, Display Layout, or Hardcopy Layout. You can install the macro script for all windows of that type or only the current one.

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

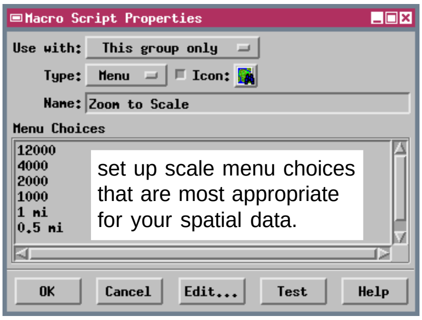
Sample Macro Script: Zoom to Scale

The Zoom to Scale macro script (ZOOMTO.SML) is one of several sample macro scripts are provided in the MACRSCR subdirectory of the SCRIPTS directory. This macro script lets the viewer redisplay the View window at one of several map scales selected from the script button's dropdown menu (or the script's submenu in the Macros menu cascade). For proper script function, the objects in the view window must be either georeferenced or scale-calibrated.

The scale 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.

STEPS
- ☑ run Display / Spatial Data
- ☑ click the Open icon button on the Display toolbar and choose Open Group from the dropdown menu
- ☑ navigate to the SML directory and select GROUPZOOMTO from the VIEWSHED Project File
- ☑ use the installed Zoom to Scale icon button on the View window to vary the zoom
- ☑ choose Options / Customize / Macro Scripts from the View window
- ☑ use the Properties and Edit icon buttons to examine the settings and script
- ☑ close the group


Macro Script Properties

Use with: This group only

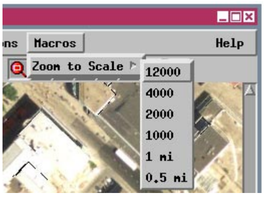
Type: Menu   Icon:

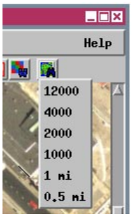
Name: Zoom to Scale

Menu Choices

```
12000
4000
2000
1000
1 mi
0.5 mi
```

set up scale menu choices that are most appropriate for your spatial data.

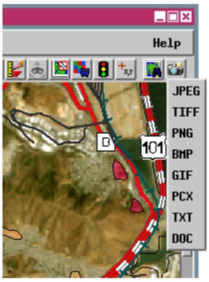




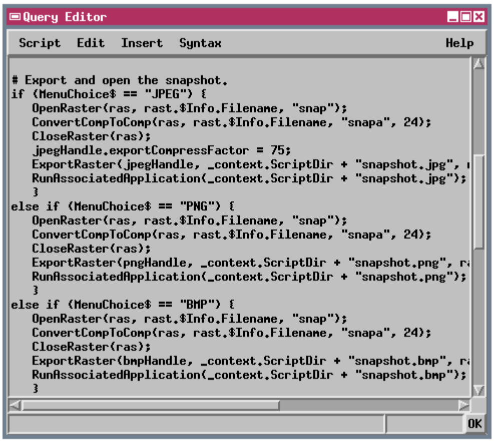
More about the Zoom to Scale macro script is available in an online document at

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.



```
# 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
   RunAssociatedApplication(_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", ra
   RunAssociatedApplication(_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", ra
   RunAssociatedApplication(_context.ScriptDir + "snapshot.bmp");
   }
```

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).

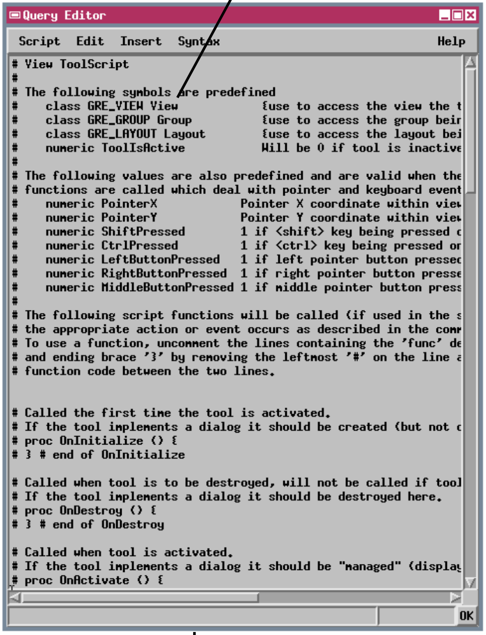
Saved snapshot of View window with raster background and several vector overlays.

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.

Tool Script Templates

Tool scripts can be run from an icon button on the View window toolbar or from the Tools menu. To add a tool script, choose Options / Customize / Tool Scripts from the View window in any TNT process that has a View window. Making this selection opens the Customize Tool Scripts window, which is nearly identical to the Customize Macro Scripts window discussed previously.

A number of class and numeric variables are predefined and available for immediate use in tool scripts.

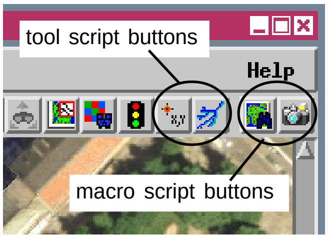


```
# View ToolScript
#
# The following symbols are predefined
#    class GRE_VIEW View            {use to access the view the t
#    class GRE_GROUP Group          {use to access the group beir
#    class GRE_LAYOUT Layout        {use to access the layout bei
#    numeric ToolIsActive           Will be 0 if tool is inactive
#
# The following values are also predefined and are valid when the
# functions are called which deal with pointer and keyboard event
#    numeric PointerX            Pointer X coordinate within view
#    numeric PointerY            Pointer Y coordinate within view
#    numeric ShiftPressed        1 if <shift> key being pressed c
#    numeric CtrlPressed         1 if <ctrl> key being pressed or
#    numeric LeftButtonPressed   1 if left pointer button pressec
#    numeric RightButtonPressed  1 if right pointer button presse
#    numeric MiddleButtonPressed 1 if middle pointer button press
#
# The following script functions will be called (if used in the s
# the appropriate action or event occurs as described in the comm
# To use a function, uncomment the lines containing the 'func' de
# and ending brace '}' by removing the leftmost '#' on the line a
# function code between the two lines.


# Called the first time the tool is activated.
# If the tool implements a dialog it should be created (but not c
# proc OnInitialize () {
# } # end of OnInitialize

# Called when tool is to be destroyed, will not be called if tool
# If the tool implements a dialog it should be destroyed here.
# proc OnDestroy () {
# } # end of OnDestroy

# Called when tool is activated.
# If the tool implements a dialog it should be "managed" (display
# proc OnActivate () {
```

To create a new tool script, click on the New icon button to open the Query Editor window, which shows the tool script template. The template lists a number of predefined symbols and values that you can use in any tool script. The predefined values include the X and Y coordinates of the screen cursor within the view (in pixels) and values that record mouse button actions.

Additionally, the tool script template includes skeletal definitions of procedures likely to be used in a tool script. These include procedures 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. To create your script, remove the comment characters (#) to the left of each procedure definition you need and add code to specify the desired action to be carried out by that procedure.

Tool Script icon buttons appear to the left of any Macro Script icon buttons on the View window toolbar.



Sample Tool Script: Select Point

NOTE: the next three exercises let you use tool scripts that were saved with a Display Group.

STEPS

☑ click the Open icon button on the Display toolbar and choose Open Group

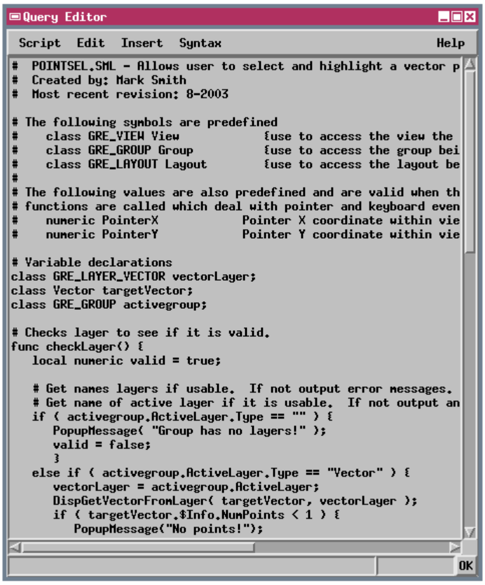
☑ choose TOOLGROUP from the TOOLS Project File in the SML directory

☑ press the Select Point toolscript icon button on the View window toolbar

☑ click [OK] in the message window, then left-click near a point in the View to select it

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() procedure. This simple script merely selects the point, but the button press procedure 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 procedures executed by mouse actions or by actions carried out in dialog windows created by the script. The definitions you provide for the predefined procedure names can call other functions and procedures that you define elsewhere in the tool script. In the point selection script, for example, the OnLeftButtonPress() procedure calls a user-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 procedure 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.



```
#  POINTSEL.SML - Allows user to select and highlight a vector p
#  Created by: Mark Smith
#  Most recent revision: 8-2003

# The following symbols are predefined
#    class GRE_VIEW View              {use to access the view the
#    class GRE_GROUP Group            {use to access the group bei
#    class GRE_LAYOUT Layout          {use to access the layout be
#
# The following values are also predefined and are valid when th
# functions are called which deal with pointer and keyboard even
#    numeric PointerX              Pointer X coordinate within vie
#    numeric PointerY              Pointer Y coordinate within vie

# Variable declarations
class GRE_LAYER_VECTOR vectorLayer;
class Vector targetVector;
class GRE_GROUP activegroup;

# Checks layer to see if it is valid.
func checkLayer() {
   local numeric 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
   if ( activegroup.ActiveLayer.Type == "" ) {
      PopupMessage( "Group has no layers!" );
      valid = false;
      }
   else if ( activegroup.ActiveLayer.Type == "Vector" ) {
      vectorLayer = activegroup.ActiveLayer;
      DispGetVectorFromLayer( targetVector, vectorLayer );
      if ( targetVector.$Info.NumPoints < 1 ) {
         PopupMessage("No points!");
```

The POINTSEL script is one of a number of sample tool scripts that are provided with the TNT products in the TOOLSCR subdirectory of the SCRIPTS directory. Other sample tool scripts in this directory are described on the following pages. You can use components from any or all of these scripts to create the custom tool you need for your specialized application.

Sample Tool Script: Select Element

The element selection script (ELEMSEL.SML) allows the user to select point, line, or polygon elements from a vector object in the View window. The selection mode is set using radio buttons on a dialog created and opened by the script. The user chooses whether a left-click of the mouse in the view selects the closest point, closest line, or enclosing polygon. The script merely selects the element, but the button press procedure could be expanded to apply further processing to or using the element.

Since a tool script creates a custom interative tool on the View window's toolbar, you can switch back and forth between the custom tool and other interactive graphic tools, such as the Zoom Box. To deal with these potential switches, tool scripts include prenamed OnInitialize() and OnActivate() procedures. The OnInitialize() procedure is called only the first time the tool is activated in a viewing session; if the script uses a custom dialog, it should be defined within this procedure (but not opened). The OnActivate() procedure is called each time the tool is activated, so the code to open the dialog window should be part of this procedure definition.

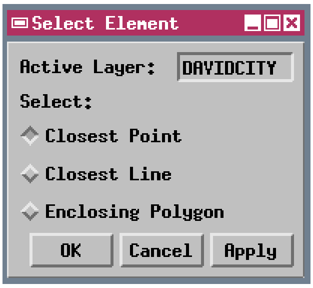
Because tool scripts use predefined variable and function names that are automatically recognized by the TNT Display process (but only in that context), script syntax should be checked only in the Query Editor window opened from the Customize Tool Scripts. The main SML editor window will return syntax errors from valid tool scripts.

Consult the Tutorial booklet *Building Dialogs in SML* for information on creating custom dialog windows.

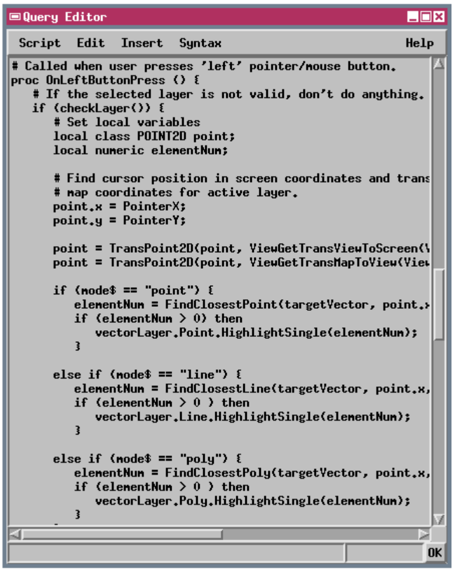
STEPS

☑ press the Select Element tool script icon button on the View window toolbar

☑ on the Select Element window that appears, use the radio buttons to choose which type of element to select



☑ left-click in the View window to select an element



```
# Called when user presses 'left' pointer/mouse button.
proc OnLeftButtonPress () {
   # If the selected layer is not valid, don't do anything.
   if (checkLayer()) {
      # Set local variables
      local class POINT2D point;
      local numeric elementNum;

      # Find cursor position in screen coordinates and trans
      # map coordinates for active layer.
      point.x = PointerX;
      point.y = PointerY;

      point = TransPoint2D(point, ViewGetTransViewToScreen(V
      point = TransPoint2D(point, ViewGetTransMapToView(View

      if (mode$ == "point") {
         elementNum = FindClosestPoint(targetVector, point.x
         if (elementNum > 0) then
            vectorLayer.Point.HighlightSingle(elementNum);
         }

      else if (mode$ == "line") {
         elementNum = FindClosestLine(targetVector, point.x,
         if (elementNum > 0 ) then
            vectorLayer.Line.HighlightSingle(elementNum);
         }

      else if (mode$ == "poly") {
         elementNum = FindClosestPoly(targetVector, point.x,
         if (elementNum > 0 ) then
            vectorLayer.Poly.HighlightSingle(elementNum);
         }
```

Modify and Extend Tool Scripts I

STEPS

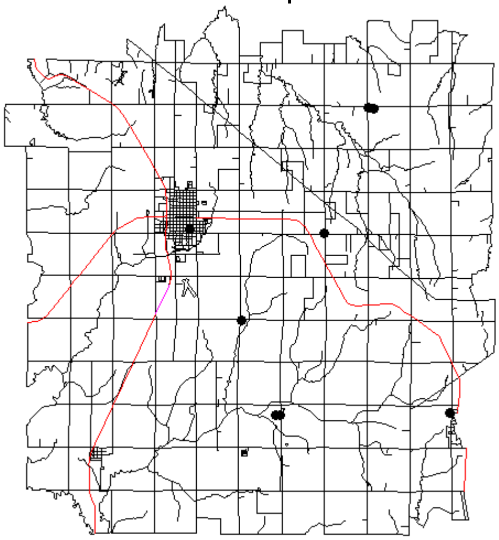
☑ press the Line by Attribute tool script icon button on the View window tool bar

☑ click [OK] in the message window, then left-click in the View window to select a line element; all lines of the same type are also selected

☑ close the Display Group when you have completed this exercise

The easiest way to develop a new tool script is to find a similar existing script (your own or one of the sample scripts described here) and modify and extend it. That's what we have done to create the tool script you use in this exercise. Script TLINBYATT.SML selects the line closest your mouse click, then highlights all lines that have the same attribute. This script was created by modifying the POINTSEL.SML tool script you used previously.

The main modifications required were in the definition of the OnLeftButtonPress() procedure, the code carried out when you press the left mouse button. This procedure is excerpted from each script on the following page. The procedure was first modified to find the closest line rather than the closest point, then to get the attribute value for that line from a particular database field and store it as a string variable. The new procedure then loops through all lines in the object to check their value for that attribute, stores the element numbers of matching lines in an array, then uses the array to highlight all of the matching lines in the View window. The procedure could be modified further to check additional attribute criteria, to extract the selected lines, or apply other processing.


All lines representing railroads selected by one mouse click next to one of the lines.

While POINTSEL.SML is generic, and will work with any vector object containing point elements, TLINBYATT.SML is tailored for a specific type of vector data: it works with any vector object imported from the U.S. Census Bureau's TIGER line files.

Modify and Extend Tool Scripts II

Excerpt from POINTSEL.SML

Procedure in POINTSEL.SML that is executed when the left mouse button is pressed. Lines in red needed to be modified to create the line selection script, and some lines were added.

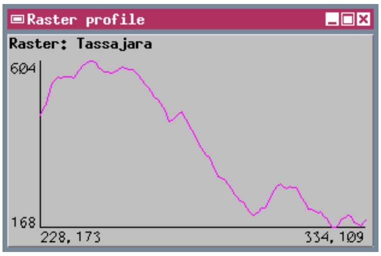
```
proc OnLeftButtonPress () {
   # If the selected layer is not valid, don't do anything.
   if ( checkLayer() ) {
      # Set local variables
      local class POINT2D point;
      local numeric elementNum;

      # Check point.
      point.x = PointerX;
      point.y = PointerY;

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

      elementNum = FindClosestPoint( targetVector, point.x, point.y, GetLastUsedGeorefObject( targetVector ) );

      if (elementNum > 0)
         vectorLayer.Point.HighlightSingle( elementNum );  # highlight single point
      }
   }  # end of OnLeftButtonPress
```

Excerpt from TLINBYATT.SML

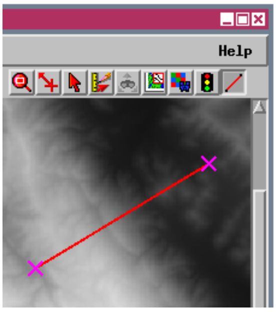
Procedure in TLINBYATT.SML that is executed when the left mouse button is pressed. Lines in blue were added and lines in red were modified to create the line selection script.

```
proc OnLeftButtonPress () {
   # If the selected layer is not valid, don't do anything.
   if ( checkLayer() ) {
      # Set local variables
      local class POINT2D point;
      local numeric elementNum;
      local string att$;
      local numeric line;
      local array numeric elemnums[0];     # array to hold element numbers of lines matching target attribute
      local numeric count = 0;

      # Check point.
      point.x = PointerX;
      point.y = PointerY;

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

      elementNum = FindClosestLine( targetVector, point.x, point.y, GetLastUsedGeorefObject( targetVector ) );

      if (elementNum > 0)
         att$ = targetVector.line[elementNum].Class_Codes.CFCC$;          # get attribute value of line

         for line = 1 to NumVectorLines( targetVector ) {                 # check attributes of all lines
            if ( targetVector.line[line].Class_Codes.CFCC$ == att$ ) {
               count += 1;                                                # if match, increase the  array size
               ResizeArrayPreserve( elemnums, count );                    # and add element number to it.
               elemnums[ count ] = line;
               }
            }
         vectorLayer.Line.HighlightMultiple( count, elemnums );   # highlight all lines with target attribute
      }
   }  # end of OnLeftButtonPress
```

Sample Tool Script: Raster Profile



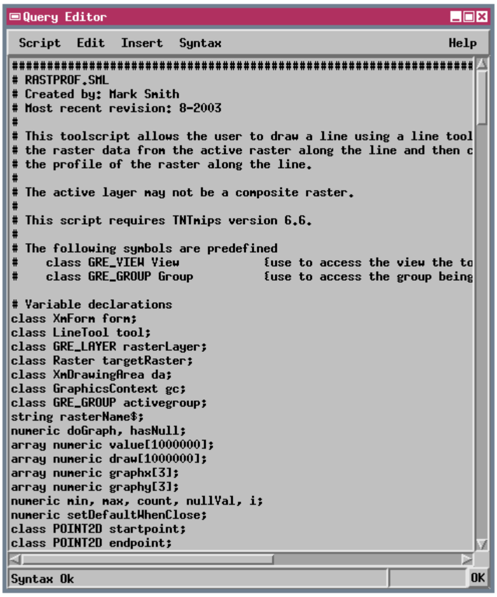
The Raster Profile tool script (RASTPROF.SML) provides a line tool that records and plots a profile of the raster cell values along a line drawn by the user. The target raster for the profile must be the active layer in the view, and x-y positions for the values are recorded in raster coordinates (column 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() procedure 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 procedure.

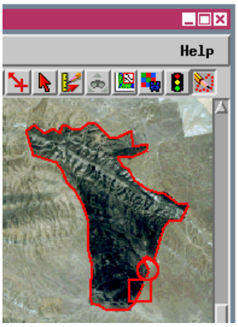


```
####################################################################
# RASTPROF.SML
# Created by: Mark Smith
# Most recent revision: 8-2003
#
# This toolscript allows the user to draw a line using a line tool
# the raster data from the active raster along the line and then c
# the profile of the raster along the line.
#
# The active layer may not be a composite raster.
#
# This script requires TNTmips version 6.6.
#
# The following symbols are predefined
#    class GRE_VIEW View            {use to access the view the to
#    class GRE_GROUP Group          {use to access the group being

# Variable declarations
class XmForm form;
class LineTool tool;
class GRE_LAYER rasterLayer;
class Raster targetRaster;
class XmDrawingArea da;
class GraphicsContext gc;
class GRE_GROUP activegroup;
string rasterName$;
numeric doGraph, hasNull;
array numeric value[1000000];
array numeric draw[1000000];
array numeric graphx[3];
array numeric graphy[3];
numeric min, max, count, nullVal, i;
numeric setDefaultWhenClose;
class POINT2D startpoint;
class POINT2D endpoint;
```

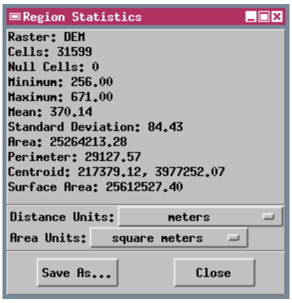
The script also demonstrates how the result of an action can be shown graphically in a window created by the script. The code that draws the graph axes, labels, and profile is contained in the procedure `cbRedraw()` defined in the script.

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.

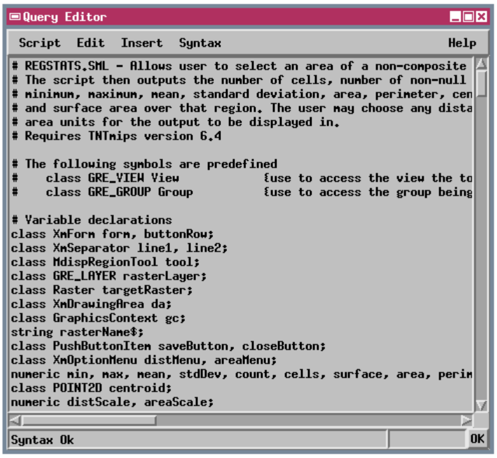


This script operates on a raster object that is the active layer in the view. In the example shown here, the polygon is drawn on an image layer overlying the active layer, which contains an elevation raster. Using the region defined by the polygon tool, the script computes the number of cells, number 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.



```
Raster: DEM
Cells: 31599
Null Cells: 0
Minimum: 256.00
Maximum: 671.00
Mean: 370.14
Standard Deviation: 84.43
Area: 25264213.28
Perimeter: 29127.57
Centroid: 217379.12, 3977252.07
Surface Area: 25612527.40
```

Distance Units: meters
Area Units: square meters



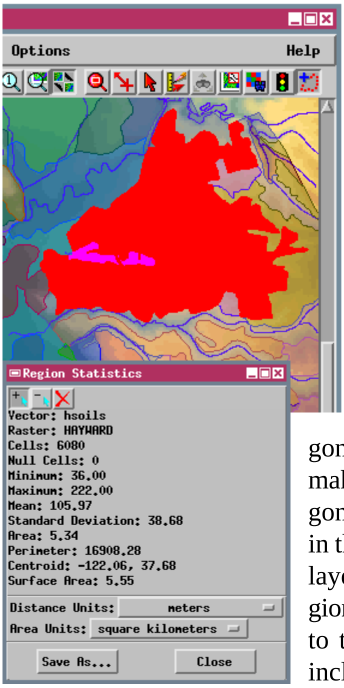
```
# REGSTATS.SML - Allows user to select an area of a non-composite
# The script then outputs the number of cells, number of non-null
# minimum, maximum, mean, standard deviation, area, perimeter, cen
# and surface area over that region. The user may choose any dista
# area units for the output to be displayed in.
# Requires TNTmips version 6.4

# The following symbols are predefined
#    class GRE_VIEW View           {use to access the view the to
#    class GRE_GROUP Group         {use to access the group being

# Variable declarations
class XmForm form, buttonRow;
class XmSeparator line1, line2;
class MdispRegionTool tool;
class GRE_LAYER rasterLayer;
class Raster targetRaster;
class XmDrawingArea da;
class GraphicsContext gc;
string rasterName$;
class PushButtonItem saveButton, closeButton;
class XmOptionMenu distMenu, areaMenu;
numeric min, max, mean, stdDev, count, cells, surface, area, perim
class POINT2D centroid;
numeric distScale, areaScale;
```

Syntax Ok

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

http://www.microimages.com/relnotes/v64/polystats.pdf

Sample Tool Script: Region Statistics



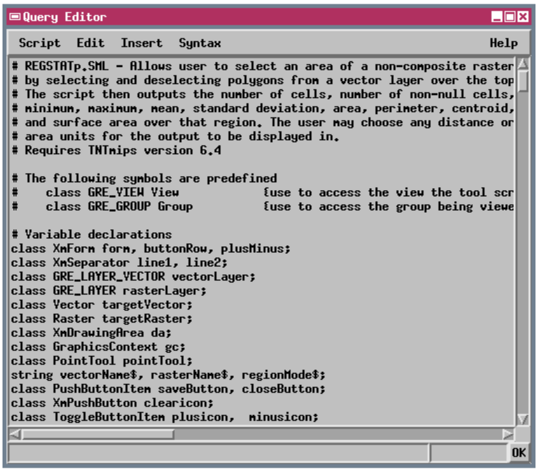
```
Vector: hsoils
Raster: HAYWARD
Cells: 6080
Null Cells: 0
Minimum: 36.00
Maximum: 222.00
Mean: 105.97
Standard Deviation: 38.68
Area: 5.34
Perimeter: 16908.28
Centroid: -122.06, 37.68
Surface Area: 5.55
```

Distance Units: meters
Area Units: square kilometers

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.

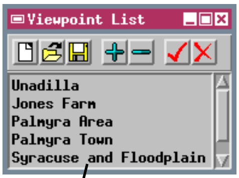


```
# REGSTATp.SML - Allows user to select an area of a non-composite raster
# by selecting and deselecting polygons from a vector layer over the top
# The script then outputs the number of cells, number of non-null cells,
# minimum, maximum, mean, standard deviation, area, perimeter, centroid,
# and surface area over that region. The user may choose any distance or
# area units for the output to be displayed in.
# Requires TNTmips version 6.4

# The following symbols are predefined
#    class GRE_VIEW View              {use to access the view the tool scr
#    class GRE_GROUP Group            {use to access the group being viewe

# Variable declarations
class XmForm form, buttonRow, plusMinus;
class XmSeparator line1, line2;
class GRE_LAYER_VECTOR vectorLayer;
class GRE_LAYER rasterLayer;
class Vector targetVector;
class Raster targetRaster;
class XmDrawingArea da;
class GraphicsContext gc;
class PointTool pointTool;
string vectorName$, rasterName$, regionMode$;
class PushButtonItem saveButton, closeButton;
class XmPushButton clearicon;
class ToggleButtonItem plusicon,  minusicon;
```

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: ViewMarks

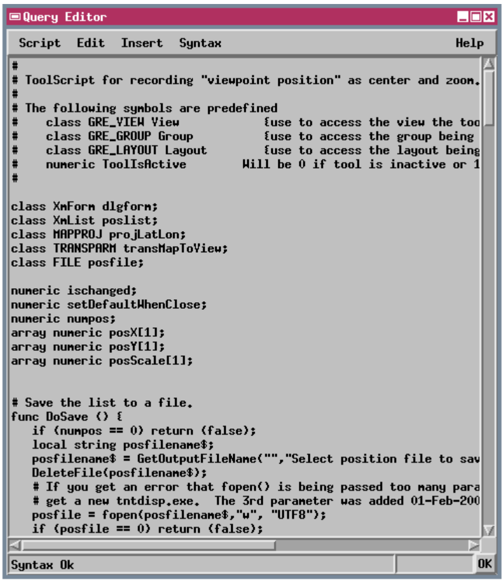
The ViewMarks tool script (VPTOOL.SML) allows you to record a list of position markers for the View window. A ViewMark records the map coordinates of the current view center (in latitude/longitude) and the map scale. Once the list is created, you can select a ViewMark and recenter the View window on 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.



```
Unadilla
Jones Farm
Palmyra Area
Palmyra Town
Syracuse and Floodplain
```

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

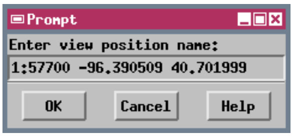
The ViewMarks script creates a Viewpoint List dialog window that provides an interactive list as well as buttons used to initiate script actions; there is no graphic tool created by the script. This dialog is created by the OnInitialize() function. The icon buttons on the window let you add or remove 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.



```
#
# ToolScript for recording "viewpoint position" as center and zoom.
#
# The following symbols are predefined
#    class GRE_VIEW View              {use to access the view the too
#    class GRE_GROUP Group            {use to access the group being
#    class GRE_LAYOUT Layout          {use to access the layout being
#    numeric ToolIsActive            Will be 0 if tool is inactive or 1
#

class XmForm dlgform;
class XmList poslist;
class MAPPROJ projLatLon;
class TRANSPARM transMapToView;
class FILE posfile;

numeric ischanged;
numeric setDefaultWhenClose;
numeric numpos;
array numeric posX[1];
array numeric posY[1];
array numeric posScale[1];


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

```
Syntax Ok
```

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 XmList). The x-coordinate, y-coordinate, and scale values are stored in separate numeric arrays.



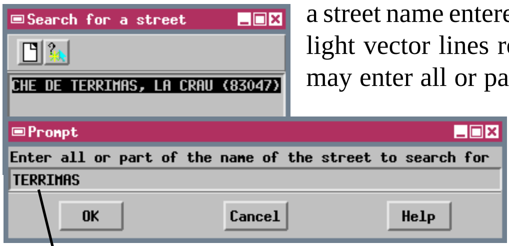
```
Enter view position name:
1:57700 -96.390509 40.701999
```

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

http://www.microimages.com/relnotes/v64/viewmarks.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 highlight 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 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.

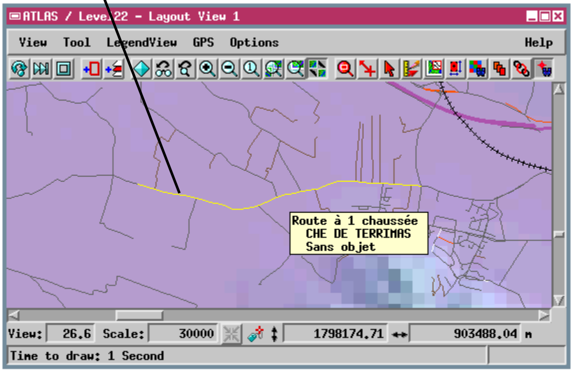


CHE DE TERRIMAS, LA CRAU (83047)

Enter all or part of the name of the street to search for

TERRIMAS

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



Route à 1 chaussée
CHE DE TERRIMAS
Sans objet

View: 26.6 Scale: 30000 1798174.71 903488.04 m

Time to draw: 1 Second

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).

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.

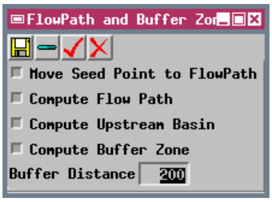
STREETS.SML is coded to work with specific geodata from a sample atlas of France. You must modify the script before it will work with other geodata and attributes.

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

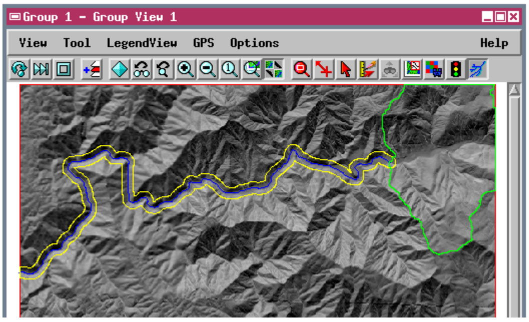
http://www.microimages.com/relnotes/v64/findstreets.pdf

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.

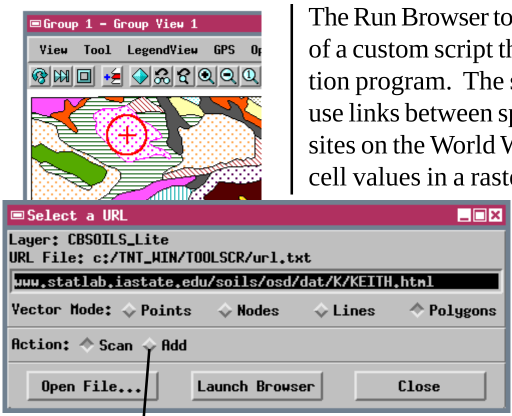
If the user intends seed points to fall along a stream course, they can turn on the Move Seed Point to Flow Path toggle button. Each seed point is then moved to the nearest precomputed flow path line before the new flow path and basin are computed. The user can place new seed points, repeat the analysis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

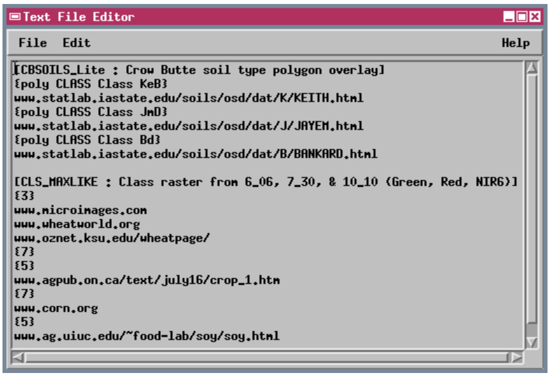


```
Layer: CBSOILS_Lite
URL File: c:/TNT_WIN/TOOLSCR/url.txt
www.statlab.iastate.edu/soils/osd/dat/K/KEITH.html
```

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].

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 that is created by the script. Choose the desired URL, then click on the Launch Browser button.



```
[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}
www.microimages.com
www.wheatworld.org
www.oznet.ksu.edu/wheatpage/
{7}
{5}
www.agpub.on.ca/text/july16/crop_1.htm
{7}
www.corn.org
{5}
www.ag.uiuc.edu/~food-lab/soy/soy.html
```

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 BEREA / BERCRPCL.RVC / CLS_MAXLIKE.

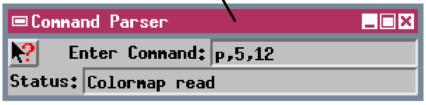
More about the Run Browser tool script is available in an online document at

http://www.microimages.com/relnotes/v64/runbrowser.pdf

Sample Tool Script: Command Parser

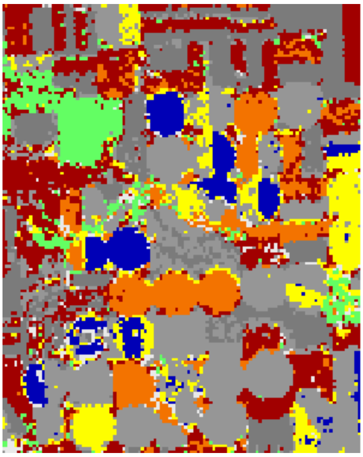
Several of the tool scripts discussed previously create a control window that allows users to execute script actions using push buttons or other graphical interface controls. The Command Parser tool script (COMPAR.SML) demonstrates a script design that creates a "command line" interface for executing 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.



Enter Command: p,5,12
Status: Colormap read

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".



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.

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.

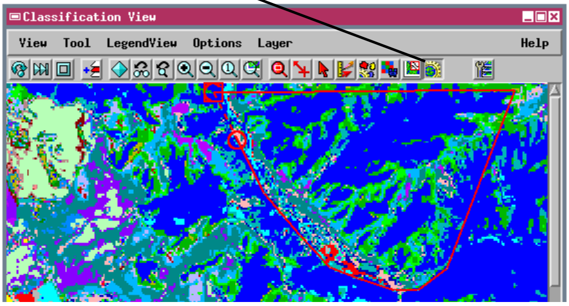
More about the Command Parser tool script is available in an online document at

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

Sample Tool Script: FRAGSTATS

If a tool script is installed for use with any 2D Group, it can be run from any view window in any TNT process. 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.

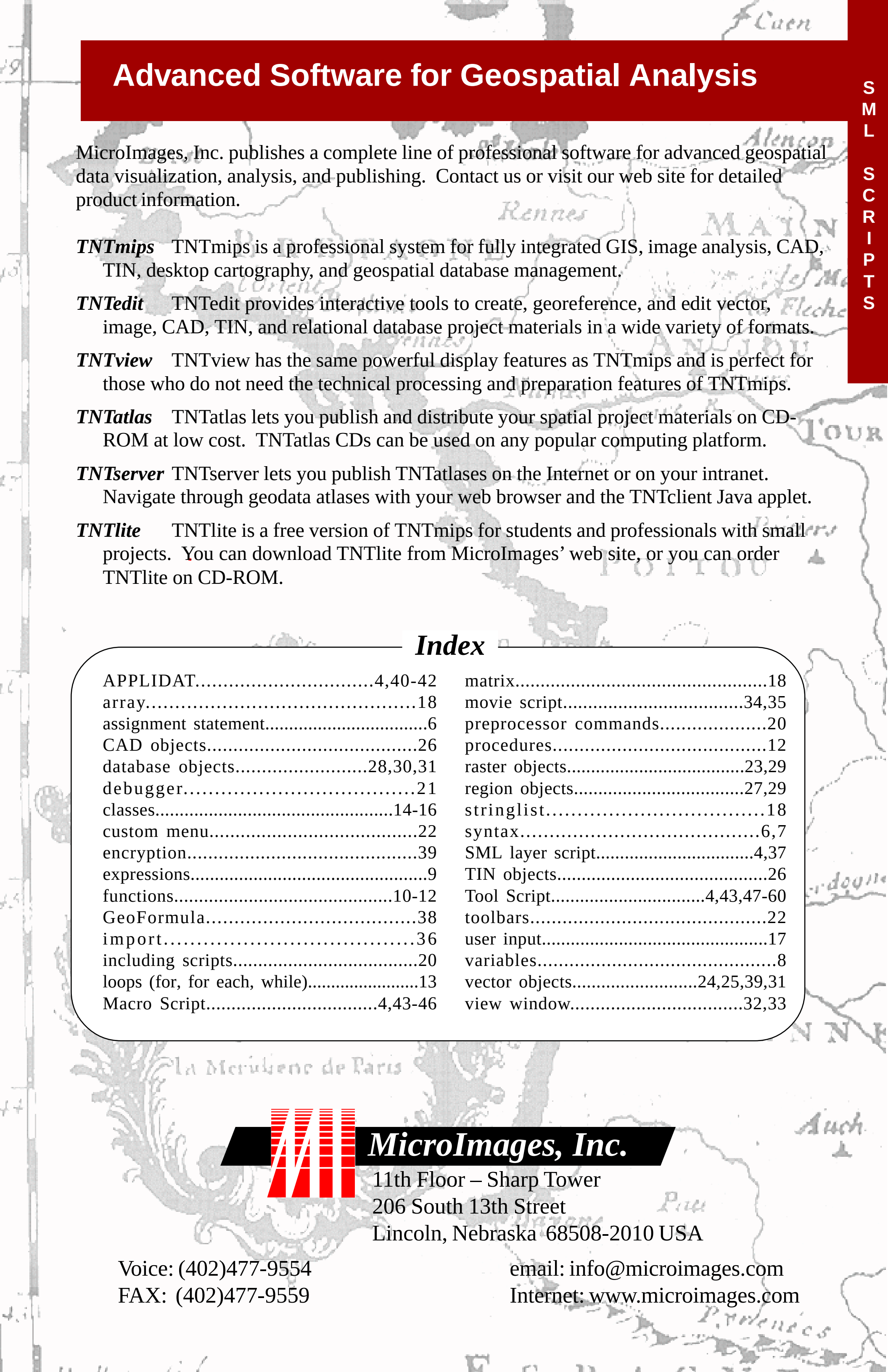


A separate script for running FRAGSTATS from the SML process interface is also available. FRAGSTAT.SML can be found in the SCRIPTS / GENERAL directory. This script requires that you provide both the class raster and a binary mask raster to define the area of interest.

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



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

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