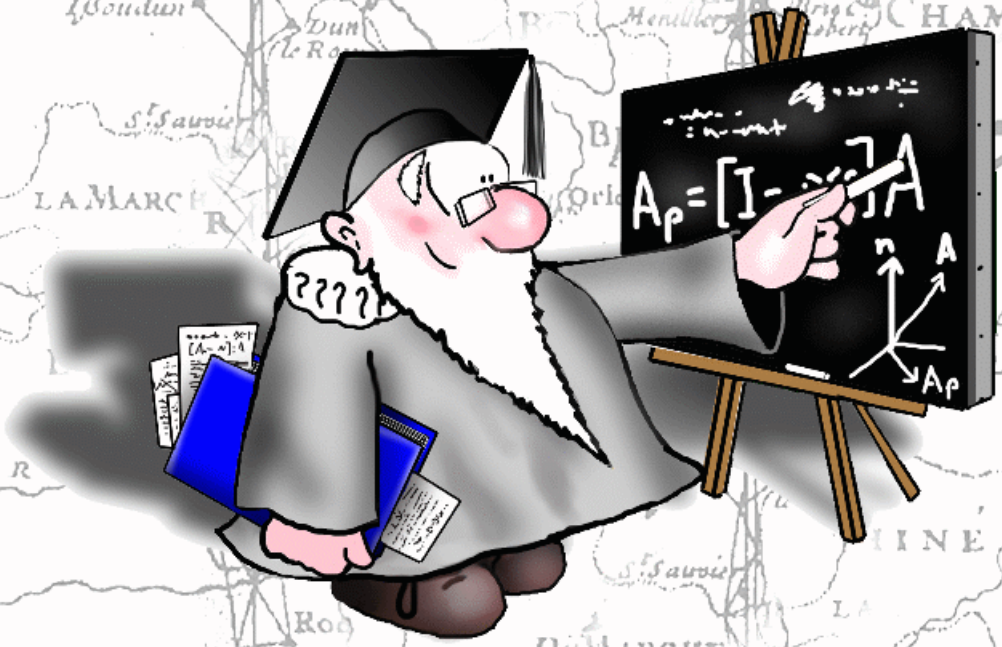
WRITING SML

Tutorial

Writing Scripts with SML



in

TNTmips®

TNTedit™

Before Getting Started

This booklet introduces the fundamentals of creating scripts in the Geospatial Scripting Language (SML) in TNTgis. 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 TNTgis Project Files.

Prerequisite Skills This booklet assumes that you have completed the exercises in the *Getting Started with TNTmips, TNTedit, and TNTview*; *Introduction to the Display Interface*; and *Introduction to Geospatial Scripting* tutorial booklets. Please consult those booklets for any review of essential skills and basic techniques you need. 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 and scripts distributed with TNTgis. If you do not have access to a TNTgis DVD, you can download the data from the MicroImages web site. This booklet uses sample data files in the SML, CB_DATA, COLOR, SF_DATA, and SURFMODL sample data directories. You will also need scripts in the sample script collection described on page 4. 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 TNTgis Geospatial Scripting Language. Descriptions of sample scripts can be found in a variety of Technical Guides that are all available from MicroImages' web site.

TNTmips Pro, Basic, and Free TNTmips comes in three versions: TNTmips Pro (which requires a software license key), low-cost TNTmips Basic, and TNTmips Free. TNTmips Basic and TNTmips Free provide nearly all the capabilities of TNTmips Pro but limit the size of the geospatial objects and attribute tables that can be used in your project. TNTmips Basic and TNTmips Free allow you to create and use scripts in the Display process (database queries, style scripts, macroscripts, and others) and the Geoformula process, but only TNTmips Pro allows you to create and run complex standalone geospatial scripts.

Randall B. Smith, Ph.D., 21 December 2015
©MicroImages, Inc., 1997—2015

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 TNTmips.

http://www.microimages.com

SML in TNTgis

The Geopatial Scripting Language (SML) is the general-purpose scripting language used throughout TNTgis. 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 TNTgis. SML has evolved as the capabilities of TNTgis has 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 standalone SML scripts in TNTmips Pro, TNTedit, and TNTscript. Scripts that run in the Display process (tool scripts, macroscripts, and display control scripts) can also be used in TNTview and distributed and used in TNTatlas. 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++) 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.

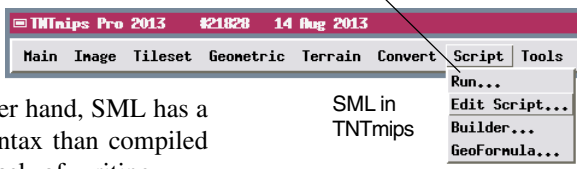
STEPS

☑ select Script / Edit Script from the TNTmips Pro menu

The exercises on pp. 4-38 introduce the SML Editor and basic SML scripting concepts and conventions. Pages 38-52 discuss specific program techniques for different types of geodata objects. The remainder of the booklet introduces advanced SML development techniques and advanced script types.

The tutorial booklet *Introduction to Scripting* provides an overview of different types of scripts that you can create and use in TNTgis.

Choose Script / Run to run a completed SML script without opening the SML Editor.



SML in TNTmips

SML in TNTedit

Sample Geospatial Scripts

A large collection of sample geospatial scripts of all types is distributed with TNTgis. If you have installed the full version of TNTgis, a file entitled SCRIPTS.ZIP should be present in your TNTgis installation directory. Unzip this file to create a SCRIPTS directory with subdirectories containing sample scripts in various categories. You can also download the latest version of this zip file from the scripting page on the MicroImages web site:

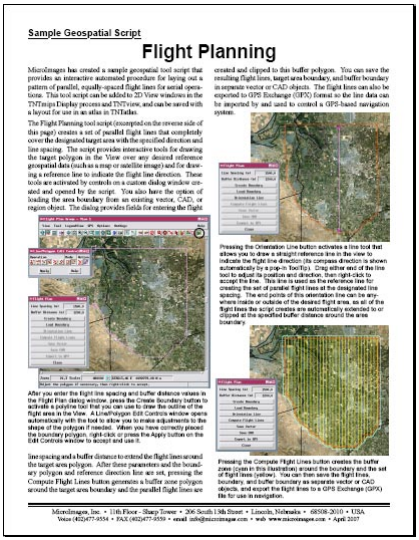
www.microimages.com/sml/index.htm

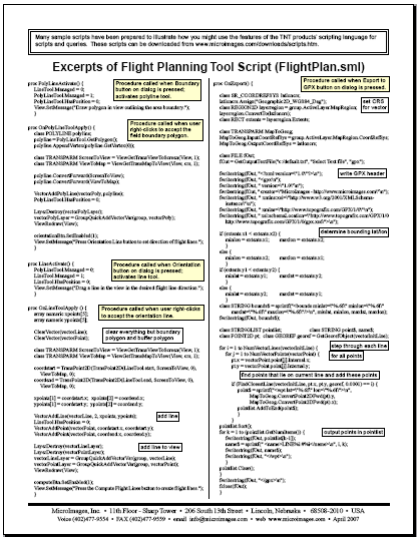
This Scripting page is also an excellent resource for the latest information about geospatial scripting in the TNT products. It provides links to pages covering different script categories and topics. These individual category pages provide examples and descriptions of different scripts in that category.

Many sample scripts are discussed in illustrated color Technical Guides that provide detailed information about the sample script along with annotated excerpts. Links to these TechGuides can be found with the script descriptions on the script category web pages. You can also access them from the Technical Guide catalog page:

www.microimages.com/documentation/html/category.htm

TechGuides illustrating sample scripts can be found under the following category links: Scripting, Scripts by Jack, CartoScripts, GraphTips, Macro Script, and Tool Script.





Run VIEWSHED.SML

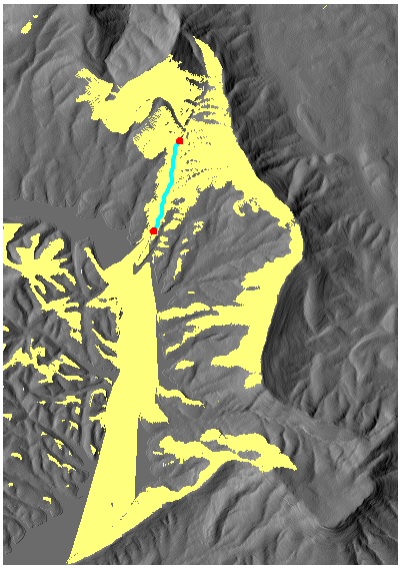
The VIEWSHED.SML script is an example of an SML process script. The script and its sample data in VIEWSHED.RVC are contained on the TNTgis DVD and are also available on the MicroImages web site. The script creates an output binary raster object that shows which parts of its input elevation surface are 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 SML Editor window 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 SML Editor 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.

The SML Editor

STEPS

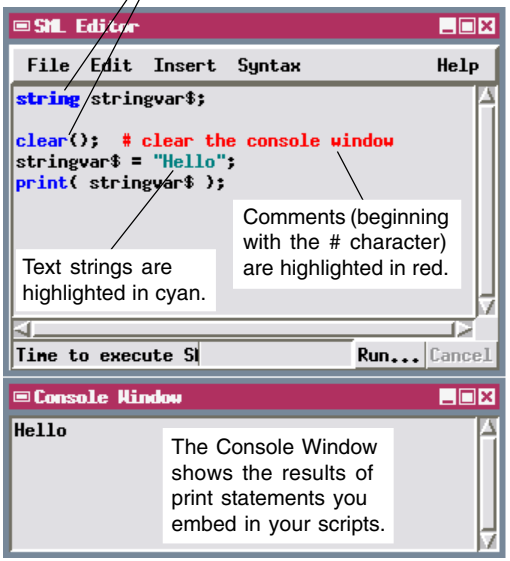
- ☑ clear the SML Editor by selecting New from the File menu
- ☑ type in the script shown in the illustration below
- ☑ click [Run] to execute the script
- ☑ choose File / Save As / *.sml File and save the script as HELLO.SML

The SML Editor window provides all of the tools required for you to create and run standalone SML scripts. To illustrate some of the basic elements of SML syntax, type in the sample script illustrated below into the SML Editor. 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. Note that the SML Editor uses color syntax highlighting to make scripts more readable.

Color syntax highlighting in the SML Editor window makes your scripts easier to read and understand.

Keywords and names of predefined functions and classes are highlighted in blue.

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.



The second program statement is followed by a comment. 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.

Checking Syntax

The Check option on the Syntax menu checks your script for syntax problems. For historical reasons SML includes two levels of syntax checking, basic and strict. Basic SML syntax rules cover proper spelling of function and class names, the number and type of function parameters, proper closing of parentheses and loops, and so on. Strict SML syntax rules have also been instituted to help ensure the correct interpretation of complex scripts:

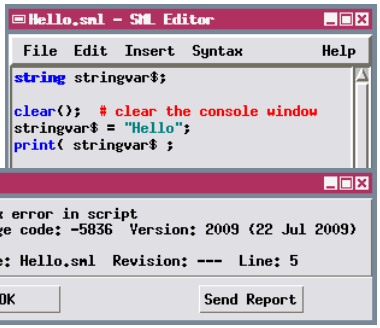
1. All variables must be declared before they are used in a statement.
2. Assigned variable values must match the declared variable type.
3. All statements must end in a semicolon.

A violation of the basic SML syntax rules is reported as an error in the message line at the bottom of the SML Editor window. The cursor is placed at the end of the last part of the script that the syntax 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. A script cannot be run until all basic syntax errors have been corrected.

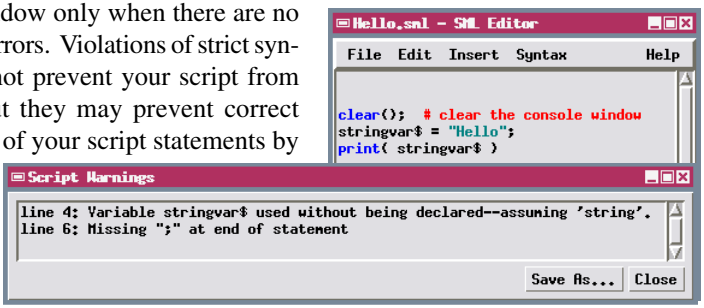
Violations of strict syntax rules are reported in a Script Warnings window only when there are no basic syntax errors. Violations of strict syntax rules do not prevent your script from being run, but they may prevent correct interpretation of your script statements by the SML parser.

STEPS

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



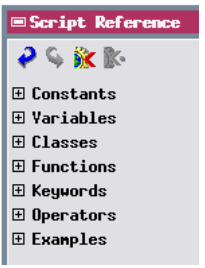
- ☑ restore the closing parenthesis to the print statement
- ☑ delete the first statement in the script (the variable declaration) and the semicolon from the end of the last statement
- ☑ select Syntax / Check from the SML window
- ☑ click [Close] on the resulting Script Warnings window



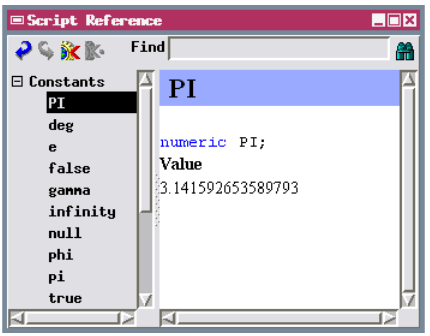
Script Reference Window

STEPS

- ☑ choose Insert / Symbol from the SML Editor window
- ☑ in the tree control in the left panel of the Script Reference window, press the plus sign icon next to Constants to show the entries in this category
- ☑ left-click on PI in the Constants list

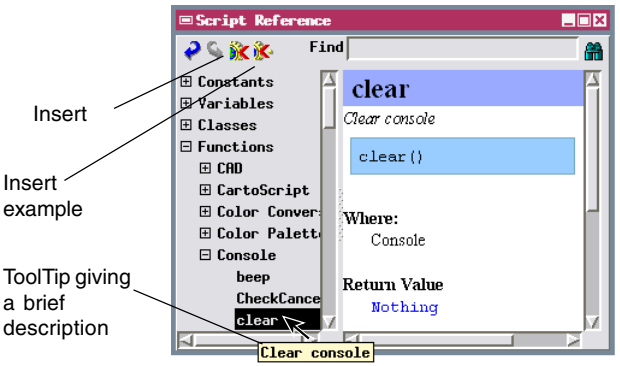


Choosing any option from the SML Editor's Insert menu opens the Script Reference window. The left side of this window is a tree control that lists the script reference categories, including Constants, Variables, Classes, Functions, and others. Click on the plus sign icon next to any of these category entries to show the listing of items in that category. Pausing the mouse over the name of a class, function, keyword, or operator shows a ToolTip with a brief description of the item. When you left-click on an individual entry in the list, the pane in the right side of the Script Reference window shows the description and further documentation for that entry. You can keep the Script Reference window open while you continue to write and edit your script.



- ☑ press the minus sign icon next to Constants to collapse this category
- ☑ expand the Functions category
- ☑ expand the Console Function group in the Functions list
- ☑ select the clear function

You can use the Insert icon button on the Script Reference window to insert any selected mathematical constant, keyword, operator, or predefined function into your script. The documentation for many of the predefined functions includes an example script. When you have selected such a function in the list, the Insert Example button becomes active, allowing you to insert the complete script example into the script you are editing.

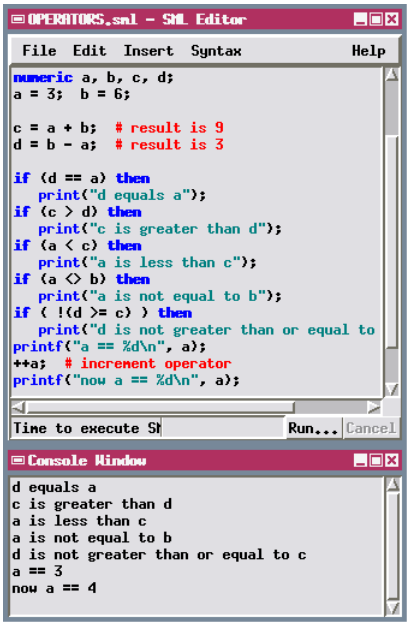


Keywords and Operators

The entries in Keywords list are reserved words that have special meaning to the SML parser. Keywords are reserved for variable types used for declarations (such as *array*, *class*, *numeric*, *raster*), conditional tests (*if*, *then*, *else, and, or*), and process flow control (*for*, *while*, *to*, *step*, *return*, *break*). There are also a number of preprocessor commands (each beginning with the "$" symbol) that are parsed before the remainder of the script to provide additional control. Preprocessor commands are discussed in more detail on a later page.

STEPS

- ☑ in the Script Reference window, expand the Operators list and examine the available operators
- ☑ choose File / Open / *.SML File and select SML / OPERATORS.SML
- ☑ examine the script, then press [Run]



The Operators list includes a variety of standard mathematical and logical operators and symbols. In addition to basic arithmetic operators (+, - , *, /), the modulo or remainder operator (%) returns the remainder of a division. The operators ++ and -- are provided as compact ways to increment or decrement (respectively) the value of a numeric variable by 1. The following two expressions are equivalent:

```
++a;    a = a + 1;
```

Likewise, the operators +=, -=, *=, /=, and %= provide shortcut forms to apply the specified mathematic operation to a numeric variable. The following two statements are equivalent:

```
a += 3;    a = a + 3;
```

Standard numerical comparison operators (> [greater than], >= [greater than or equal to], etc.) are also provided. An expression such as *a* > *b* returns either 1 (true) or 0 (false). The general comparison operator (*a* <=> *b*) returns -1 if a < b, 0 if a == b, or 1 if a > b. The logical operators *and* and *or* can be represented either by keyword or by their operator symbol (&& and ||).

Variables

STEPS
- ☑ choose 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]

NOTE: you can declare variable names for spatial objects using the keywords raster, vector, cad, tin, and region. However, using the more modern class structures for spatial objects in SML is preferred and will provide better integration with other SML classes.

You can declare and use *variables* for string, numeric, logical, array, and class entities in SML. Simple variables are also available for spatial objects (raster, vector, CAD, TIN, and region). However, these and other spatial objects are now also represented by classes that provide greater control and flexibility.

Variables in SML follow these conventions:

String: values in assignment statements must be enclosed in double or single quotes. Single quotes must be used to enclose strings with embedded carriage returns (multi-line strings).

Numeric: values can be integer or decimal.

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: numeric arrays are implemented as a variable type and can be either one-dimensional or two-dimensional. Your declaration of an array must specify the name of the array and the size of each dimension of the array as follows:

```
array numeric arrayName[10];
array numeric arrayName[4,5];
```

In the declaration of a two-dimensional array, the first value specifies the number of columns and the second number specifies the number of rows.

Class: declarations of class variables start with the "class" keyword followed by the class name and the name of the class instance:

```
class COLOR red;
class STRING name;
```

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 placed 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 *then* keyword is optional, as is the *else* clause:

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

Conditional statements can test for logical combinations of conditions as well:

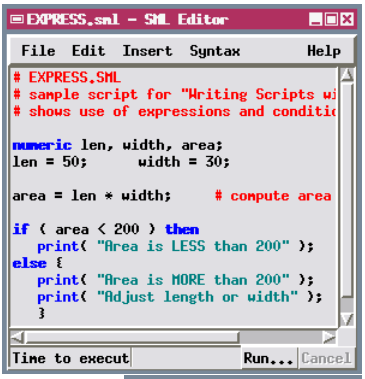
```
if (<condition1> and <condition2>) then
<statement>;
if (<condition1> or <condition2>) then
<statement>;
```

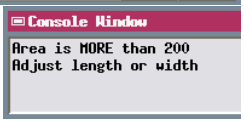
A **complex statement** involves multiple actions on separate script lines and is bracketed by the keywords *begin* and *end* in the form

```
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;
}
```





STEPS

- ☑ choose File / Open / *.SML File and select SML / EXPRESS.SML
- ☑ run the script
- ☑ decrease the value for the *width* variable and run the script again

A single equal sign (=) is the assignment operator used to assign a value to a variable:

```
numeric num = 5;
```

To test for equality in a conditional expression, use the Equal To comparison operator, a double equal sign (==) as in:

```
if (num === 5) {
  <statement>;
}
```

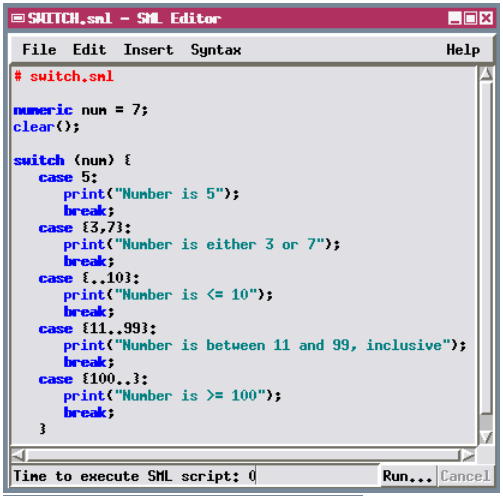
Branching Using Switch and Case

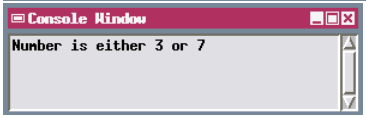
STEPS

- ☑ choose File / Open / *.SML File and select SML / SWITCH.SML
- ☑ run the script
- ☑ change the value of variable *num* and run again

Process branching based on the value of a numeric or string variable can be set up using a series of if .. then .. else statements:

```
if (num == 5) then
  <statement>;
else if (num == 3 or num == 7) then
  <statement>;
else if (num >= 11 and num <= 99) then
  <statement>;
```





A switch construct using a string expression would have the form:

```
switch (outformat$) {
  case "GeoJP2":
    <statement>;
    <statement>;
    ...
    break;
  case "GeoTIFF":
    <statement>;
    <statement>;
    ...
    break;
  }
```

For instances where multiple values and outcomes need to be specified, this structure can be cumbersome. The **switch** construct provides a more compact syntax.

A switch statement can check the value of either an integer or string expression (enclosed in parentheses following the *switch* keyword). The values to be tested are specified using the *case* keyword. Case statements can specify a single value, a series of values, inequality to a value, or a range of numeric values. Case statements use a very compact syntax, as illustrated by the sample script in this exercise. Note that each *case* value specification is followed by a colon (:). Multiple statements can be executed for each case without requiring enclosure between *begin-end* keywords or braces. The series of statements for each case are terminated by a statement with the *break* keyword. The entire set of cases must be enclosed within braces or *begin-end* keywords.

Raster Algebra and "For Each" Loops

Implied Loops. For raster objects you can use simple "raster algebra" in an assignment statement to execute an implied loop through all of the cells in the raster. The expression on the right side of the statement is evaluated for each cell and the result assigned to each cell in the raster on the left side:

```
R = R * scale; # rescale each cell in R
Rout = Rin;  # copy values from Rin to Rout
```

Rasters objects that you use in implied loops must have the same line-column dimensions.

For each statements explicitly loop through all cells of the specified raster. These statements have the forms:

```
for each Rastvar {<statements>}
for each Rastvar[lin,col] <statements>
```

The `lin` and `col` variables in the second example can be used to indicate the line number and column number of the "current position" in the raster if these values need to be accessed within the processing loop.

You can also reference a region object in a "for each" loop to restrict processing to the cells occurring within the region:

```
for each Rastvar in Region
    <statement>
```

You can also use a "for each" loop to iterate through all of the elements of a particular type in a vector object:

```
for each vector_element[n] in V
    <statement>
```

In the vector notation, vector_element can be "point", "line", "poly", or "node". The [n] is optional and can be omitted.

```
for each poly in Vect <statement>
```

If given, the variable n is used as the loop counter.

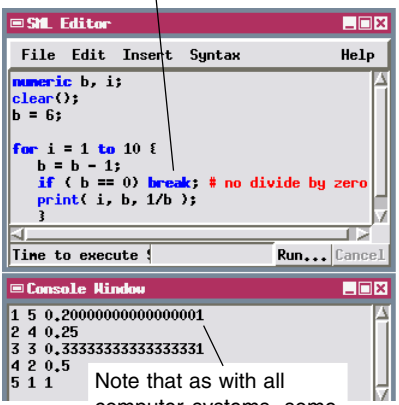
STEPS

- ☑ choose File / Open and select BREAK.SML from the SML directory
- ☑ run the script

The syntax discussed here can be used for raster objects represented by a simple raster variable, class RASTER, or class RVC_RASTER.

NOTE: the "for each" keyword sequence also may be written as one word: "foreach". 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.



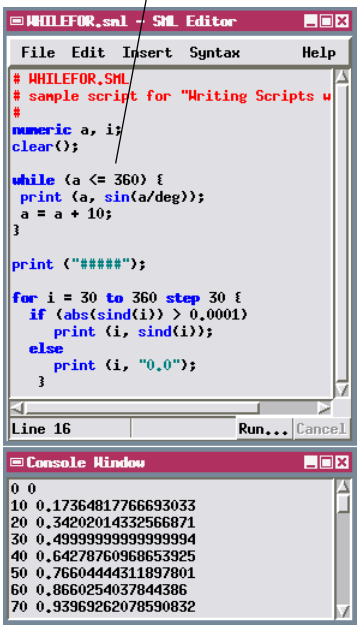
Note that as with all computer systems, some operations yield very small rounding errors in floating point values.

"For" and "While" Loops

STEPS

- ☑ choose File / Open / *.SML File and select SML / WHILEFOR.SML
- ☑ 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

The "while" loop in this example executes until variable "a" exceeds a value of 360.



The second part of the sample script uses a "for" loop to print the sines of angles from 0 to 360 degrees with a step interval of 30 degrees.

For statements have several forms:

```
for i = 1 to 11 { <statements> }
for i = 1 to 11 step 2 ( <statements> }
```

You can also use syntax as in the C programming language:

```
for (i = 1; i<= 11) ( <statements> }
for (i = 1; i <= 11; i += 2) {
    <statements> }
```

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:

```
for i = 1 to Vect.$Info.NumPoints {
        <statement; statement; ...>
        }
```

The "step" keyword (or the equivalent in C syntax) can be used if you don't need to operate on every item in the specified range. The nested loops in the example below would operate on raster cells at step intervals of 3 lines and 3 columns:

```
for i = 1 to NumLins(Rin) step 3 {
  for j = 1 to NumCols(Rin) step 3 {
    Rout = Rout + FocalMean(Rin,3,3);
    }
  }
```

Note that the "raster algebra" assignment statement in the example above does NOT invoke an implied loop, since this statement is already being executed within an explicit processing loop.

While. A"while" loop runs continuously as long as the loop condition tests true.

```
while (condition) <statement>
```

You need to be careful in using "while" loops. If the condition never becomes false, you get an infinite loop.

Built-In Functions

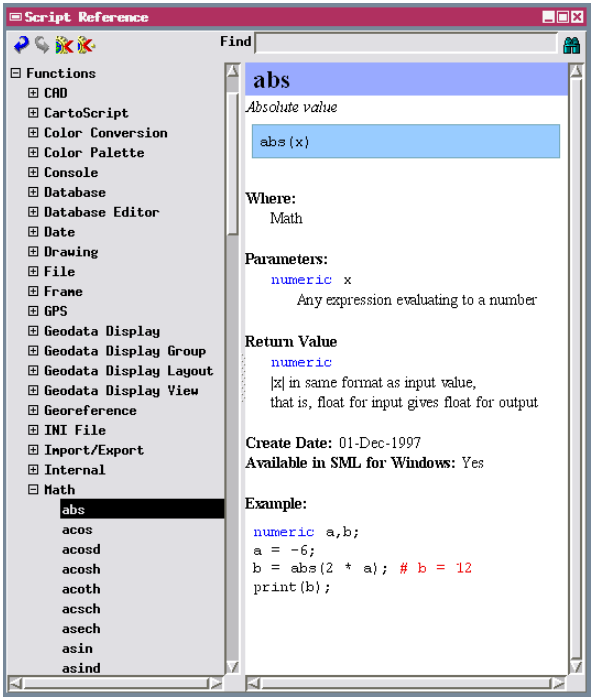
The real power of SML lies in its rich collection of built-in functions and classes. The built-in functions are organized into different function groups, each with its own expandable listing in the Functions category in the Script Reference window's tree control.

SML includes many predefined functions that allow your scripts to create, read, process, and write geospatial objects and associated database information in your TNT Project Files. Many SML functions directly utilize the compiled internal functions used in corresponding TNTmips processes, providing fast execution. Specialized functions are also provided for user input and working with displays of geospatial data. In addition, general-purpose functions are provided for standard mathematical operations, set operations, and reading from and writing to text files.

When you select a specific function in the listing, its documentation is shown automatically. The entry includes the function parameters, the value returned by the function, and in most cases a script example utilizing the function. Function names are case-sensitive.

STEPS

- ☑ clear the SML Editor with File / New
- ☑ if the Script Reference Window is not open, choose Insert / Function from the SML Editor
- ☑ expand the Functions listing in the Script Reference window
- ☑ expand the Math function listing in the Script Reference window
- ☑ select the *abs* function in the list



Keep the Script Reference window open for the next series of exercises.

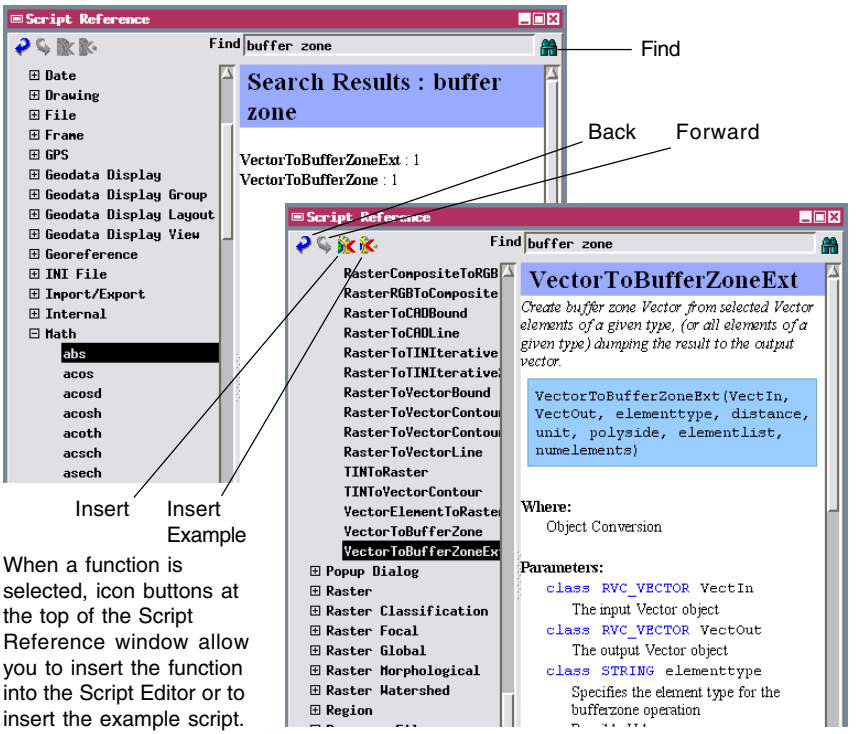
Navigating the Function List

STEPS

- ☑ type "buffer zone" into the Find field at the top of the Script Reference window
- ☑ press the Find icon button
- ☑ in the Search Results list, click on the link to VectorToBufferZoneExt
- ☑ press the Insert Example icon button
- ☑ examine the newly-inserted script lines in the SML Editor
- ☑ press the Back icon button twice to return to the abs function documentation

SML includes hundreds of built-in functions and classes, so the Script Reference window allows you to search by keywords. In this example you are looking for a function to compute a buffer zone for vector elements. A search on the words "buffer zone" locates two versions of the function in the Object Conversion function group. Selecting one of these search results takes you to the documentation for that function. For functions that include classes as parameters, the class names in the parameter documentation (shown in blue) are links to the class documentation.

Once you have followed one or more such links, you can easily navigate through the documentation locations you have visited using the Back and Forward icon buttons.



When a function is selected, icon buttons at the top of the Script Reference window allow you to insert the function into the Script Editor or to insert the example script.

Console Functions

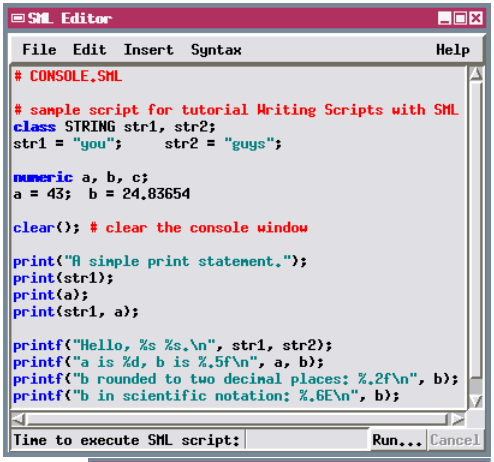
The Console function group includes functions that allow you print information to the SML Editor's Console Window. You can use print statements to provide feedback to the user of the script and to monitor the value of variables as a means of verifying correct execution as you write your script.

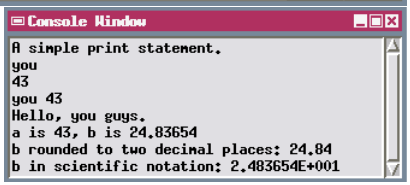
STEPS

- ☑ choose File / Open / *.SML File and select SML / CONSOLE.SML
- ☑ in the Script Reference window, expand the Console function group
- ☑ select the printf function and examine its documentation
- ☑ run the script

The print() function prints unformatted text to the console. It takes one or more values separated by commas. The parameter values can be simple text enclosed in quotes, string variables, or numeric variables. The print() function automatically starts a new output line.

The printf() function prints formatted text to the console. The syntax used for this function follows that of the printf() function in the C programming language. The list of values you supply to the function is preceded by a format specification string that determines how each following value is formatted as it is written to the output. The string consists of a set of format specification fields (one for each value passed to the function), each of which begins with a percent sign. In the output each specification field is replaced by the formatted result of the corresponding parameter value in the list. The format specification string can also contain ordinary text characters, which appear in the output as written. The last part of each specification is a conversion character that indicates the type of value and its notation. The sample script uses the following common conversion characters:





The printf() function does not automatically start a new output; you must specify a carriage return explicitly using the \n code within the format specification string.

```
s: string                    f: floating point number
d: signed integer            E: number in scientific notation
```

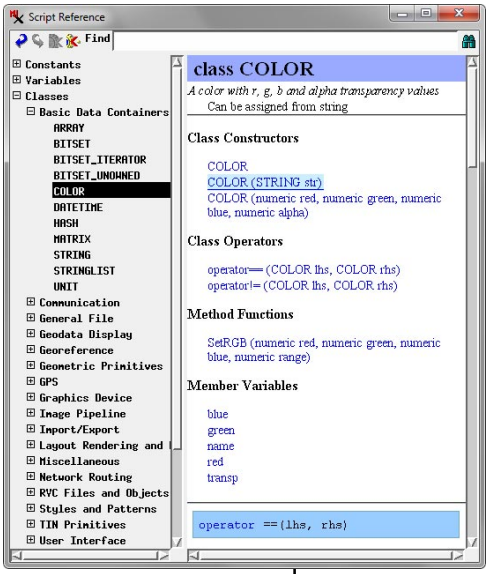
Using Classes

STEPS
- ☑ scroll up in the Script Reference window's category list to show the Classes entry
- ☑ expand the Classes entry
- ☑ expand the Basic Data Containers entry
- ☑ select class COLOR in the list

A *class* is a structure that bundles related data values and can also include functions to manipulate those data values. Class data variables are called *members*. Class functions are called *methods*.

A class may have any number of members, and the members may be of any data type, including other classes. A class is declared using the keyword "class" followed by the class name and your name for the class "instance" you are declaring:

```
class COLOR background;
```



The statement above declares an instance of class COLOR named "background". Class and member names are not case-sensitive, but your class instance names *are* case-sensitive.

Once you have defined a class instance, use the form instance_name.member to reference a member value. For example, class COLOR has four numeric members that can be assigned values in the form:

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

A class may also define operators that you can use to compare instances of that class. The COLOR class includes the "==" and "!=" operators that you can use to test whether two COLOR class instances are the same color or different colors.

The "name" member of class COLOR is a STRING class instance. It is used to set the other color values using the name of a color as defined in the RGB.txt reference file supplied with TNTmips:

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

The "name" member is write-only. Some classes may include read-only members whose values were set when the class instance was created by some other function or class. Values of read-only class members cannot be changed in your script.

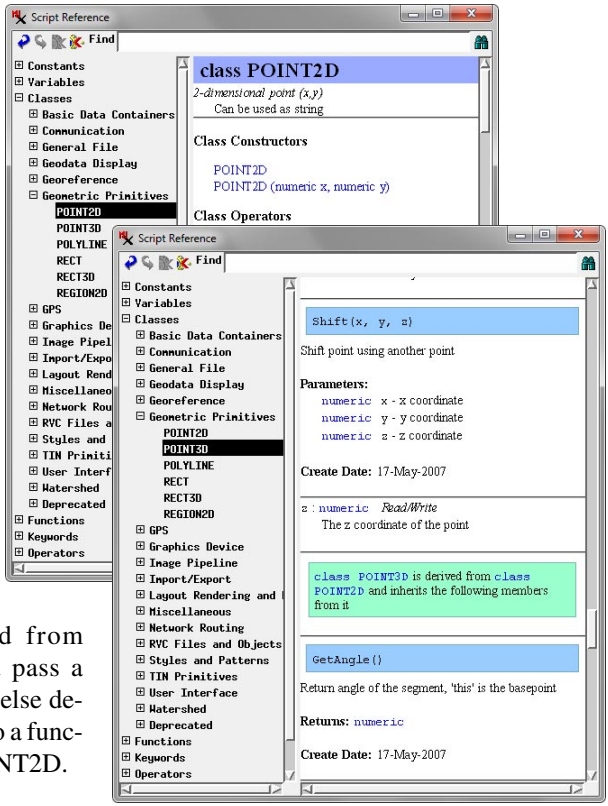
Member Inheritance and Type Checking

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 represents the location of a 3-dimensional point. Since both types of point have *x* and *y* coordinates, class POINT3D is *derived* from class POINT2D. Class POINT3D *inherits* the *x* and *y* members it has in common with POINT2D and also has its own unique member *z*. POINT3D also inherits some class methods for manipulating 2D point data from POINT2D, as well as having several methods for 3D point data. You can use inherited members and methods of a class in the same way you would its native members.

STEPS

- ☑ expand the Geometric Primitives grouping in the class list and select POINT2D and examine its members
- ☑ select POINT3D in the same group and examine its members
- ☑ scroll down in the POINT3D documentation to find the members inherited from POINT2D

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 class instance (or derivative class). But the function will refuse any variable that is not a POINT2D. For example, you could not pass such a function a COLOR class, because COLOR is not a POINT2D. By contrast, since POINT3D is derived from POINT2D, you could pass a POINT3D or anything else derived from POINT2D to a function that requires a POINT2D.



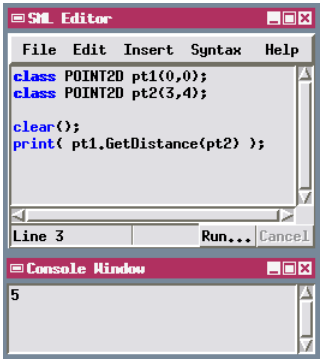
Class Methods

STEPS

- ☑ select POINT2D in the Class list
- ☑ in the listing of POINT2D class members, click on GetDistance
- ☑ examine the documentation for this method

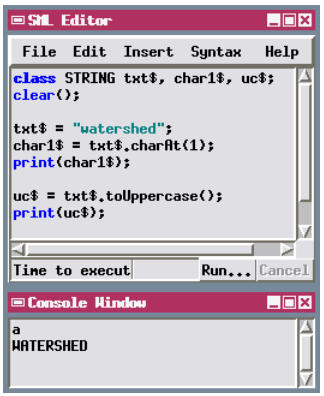
Class methods may be used to get values from or assign values to the members of the class or to perform some operation using the data contained in the class. You invoke a class method using the form instance_name.method().

Many classes have one or more special methods called constructors that let you declare the class instance and assign member values in one step. In the sample script shown to the left, you use a constructor for class POINT2D that lets you assign the values of the *x* and *y* members. Class POINT2D also includes methods to get the distance, squared distance, and angle between the current point and a second specified point. Each of these class methods returns a numeric value. Class POINT3D inherits these three methods (which take a POINT2D as the parameter) from its parent class POINT2D. In addition, in POINT3D you can pass another POINT3D to the distance functions to return the 3D distance (or squared distance) between the 3D points.



- ☑ select STRING from the Basic Data Containers group in the Class list and examine its member list
- ☑ choose File/New in the SML Editor
- ☑ enter and run the script shown below

The STRING class includes methods to return a character or larger portion from the string, to remove or replace a portion of the string, to compare the string with another, and to transform the case of the string.




```
clear();
class STRING txt$, char1$, uc$;
txt$ = "watershed";

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

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

In this example, the charAt(n) method returns the n'th character in the string (indexed with the leftmost character at index 0). The toUppercase() method returns a copy of the string in all uppercase characters

RVC System Classes

SML includes several generations of variables for working with spatial objects. The original generation is represented in the Script Reference keyword list by keywords for the spatial object variables *raster*, *vector*, *cad*, and *tin*. The next generation is a set of classes corresponding to these spatial objects: class RASTER, class VECTOR, etc. Most of these simple object classes are now deprecated but are currently still supported to allow older SML scripts to run. (Deprecated classes are shown in the Script Reference window in a category of that name.)

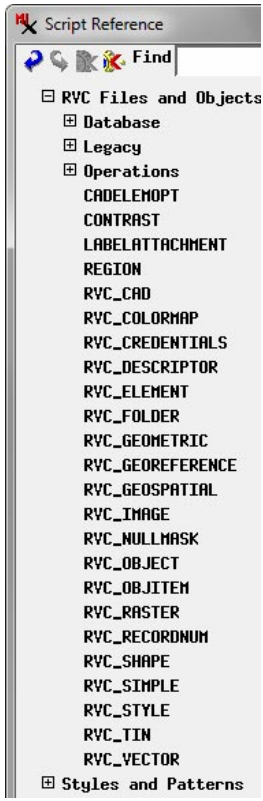
The modern classes representing spatial objects are found in the RVC System class group. These SML classes largely mirror the class structure now used in the internal program code for the TNT products. In addition to classes for the four types of spatial objects previously supported (RVC_CAD, RVC_RASTER, RVC_VECTOR, and RVC_TIN), support is provided for accessing shape objects in SML via the RVC_SHAPE class.

In addition to these spatial object classes, there are RVC classes for certain subobjects of geospatial objects, such as georeference (RVC_GEOREFERENCE), color palettes (RVC_COLORMAP) and null masks (RVC_NULLMASK) for raster objects, and style objects (RVC_STYLE) for geometric objects. This class group also includes the base classes from which the spatial object classes are derived: RVC_OBJECT, RVC_IMAGE, and RVC_GEOMETRIC. The only one of these base classes that you are likely to use directly is RVC_OBJECT, which can be used to reference or create Project Files (permanent or temporary).

The RVC_DESCRIPTOR class is a container for the name and description of an object or Project File. The RVC_OBJITEM class is used to specify a particular RVC object in a particular Project File.

STEPS

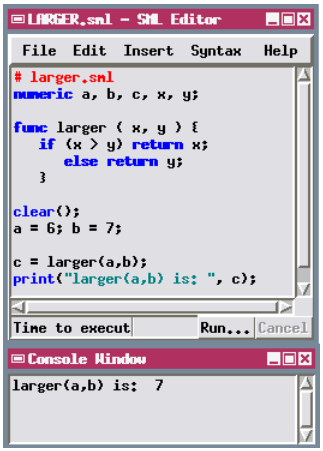
☑ in the Classes list in the Script Reference window, expand the RVC Files and Objects class group



Some newer classes that operate on spatial objects, such as the Import/Export and Image Pipeline classes, use RVC_OBJITEM class instances rather than spatial object classes to reference the objects being processed or created (more on this later).

User-Defined Functions and Procedures

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



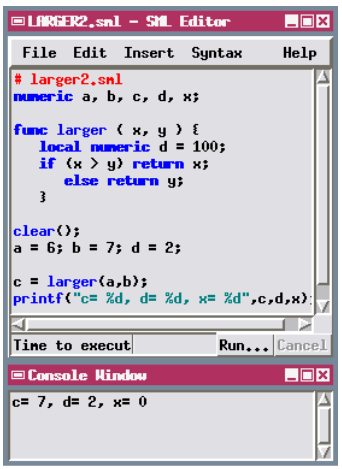
SML allows you to define your own functions and procedures that you can use to encapsulate sequences of program steps that must be repeated in several places in the script. User-defined functions return a value using the *return* keyword, whereas procedures do not. Of course you must declare a function 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. 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;  ...
```

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



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 on this page, variable *x* retains the default value 0 in the main script because the function parameter *x* is automatically local. In the second example 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.

Function Return Values and Parameters

A user-defined function returns a numeric value by default. But you can write a user-defined function to return another variable or class type by specifying the return variable type in the function definition immediately after the keyword *func*:

```
func string convertStr()
func class POINT3D convPt()
```

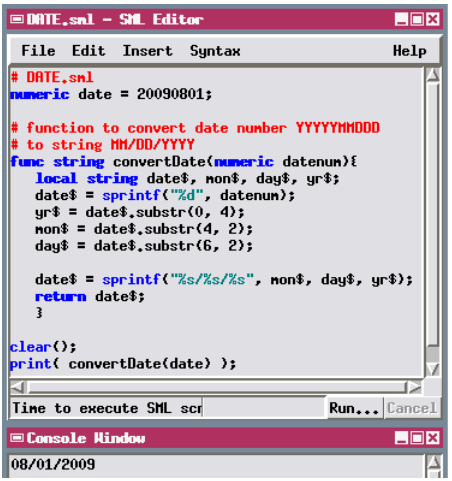
Note that you specify the variable type (including the class keyword for class variables), not the variable name you use in the function definition. You do not need to explicitly specify the return variable type for a function that returns a numeric value.

A parameters list is optional for user-defined functions and procedures. When defining or calling a function without a parameter list, simply use an empty pair of parentheses following the name. If you are using new (rather than previously-declared) variable names in the parameter list of a user-defined function or procedure (recommended), the variable type should be declared within the parameter list, as shown by the two script examples in the illustrations on this page.

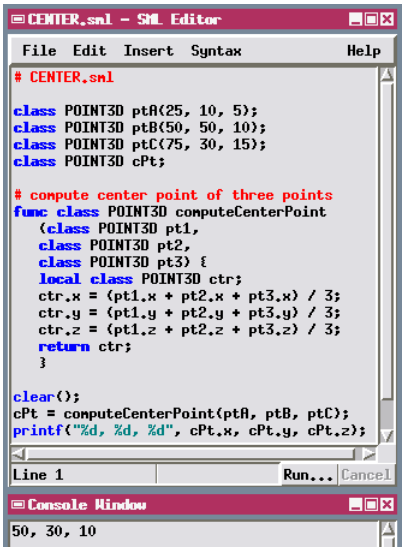
The first example script defines and uses a function to convert a numeric value representing a date (using the format YYYYMMDD) to a string with the date in the form MM/DD/YY. The second example script defines and uses a function to compute the 3D coordinates of a point that is the center of a triangle. Each input triangle vertex is a POINT3D and the function returns a POINT3D as well.

STEPS

- ☑ choose File / Open / *.SML File and open DATE.SML from the SML directory
- ☑ run the script

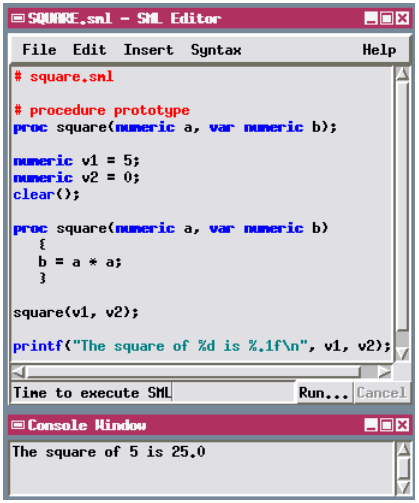


- ☑ repeat with CENTER.SML from the SML directory



Variables by Reference

STEPS
- ☑ choose File / Open and select SQUARE.SML from the SML directory
- ☑ run the script



We saw earlier that variables passed to a function or procedure are normally local in scope, as the procedure works only with a local copy of the variable. However, you can implement a procedure or function to change the value of one or more variables that are passed to it. This is done by passing to the procedure a *reference* to the original variable rather than the current value of the variable. A variable reference is indicated in the parameter list by declaring the variable in the list using the keyword *var*, as in the Pascal programming language.

In the *square* procedure defined in the script example, the first parameter is a local numeric parameter (the value to be squared). The second parameter is a variable reference declaring the variable in which to store the squared value. When the procedure is called, global variable v2 is passed to the procedure as the second variable; the procedure changes its value from 0 to 25. You can pass multiple variables by reference to a function or procedure to set multiple variable values within the same function/procedure.

A complex processing script may include a number of related and interdependent functions and procedures (e.g., one function calls one or more of the other related functions). Such interdependencies may make it difficult to define every function/procedure in the script before it is referenced by one of the other functions/procedures. To avoid this problem, you can list function/procedure *prototypes* (name and parameter list without the definition) near the top of the script, before any of the functions/procedures are called or defined. A procedure prototype is included for the *square* procedure at the top of the example script.

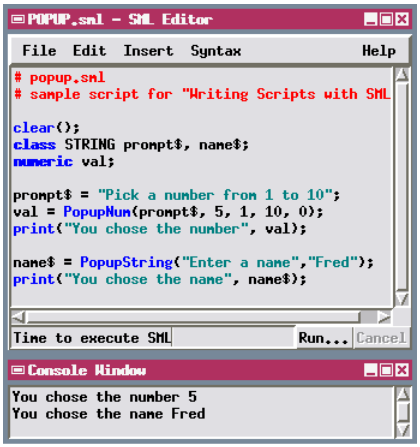
User-defined functions cannot return class instances that represent spatial objects (e.g. RVC_RASTER, RVC_SHAPE, etc.). You can, however, pass spatial object variables by reference to procedures or functions for modification.

At this time you cannot declare function prototypes for functions that return a class variable.

Interactive User Input

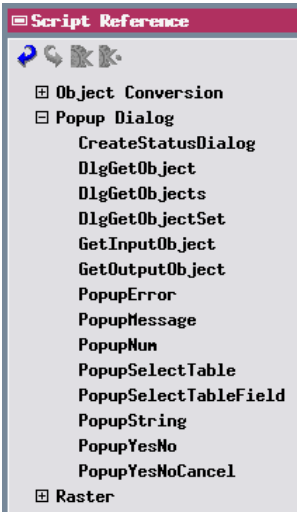
You have several options for providing interactive user input to your scripts. The simplest method for the script writer is to use popup dialog functions. SML includes predefined functions in the Popup Dialog function group that open dialogs for input of numeric or string values, yes-no responses, and displaying messages and errors. This function group also includes functions to allow selection of a database table or table and field. 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.

STEPS
- ☑ in the Script Reference window expand the Functions list and then the Popup Dialog list
- ☑ choose File / Open / *.SML File and open POPUP.SML from the SML directory
- ☑ run the script

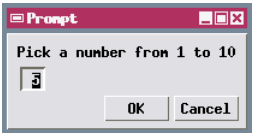


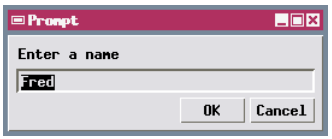
The Raster, Vector, CAD, and TIN function groups include predefined functions for selecting an input and output object of that type (such as the GetInputRaster and GetOutputRaster functions in the Raster function group). These functions take an object variable in the appropriate object class for the selected object (such as class RVC_RASTER for a selected raster object).

The File function group provides predefined functions that open dialogs for selecting a directory, input and output file names and text file names.



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

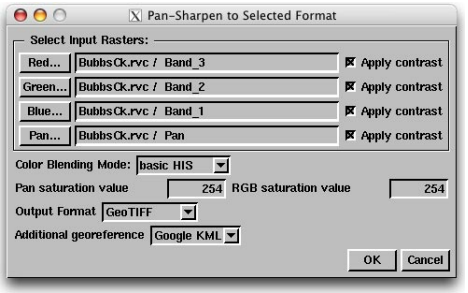




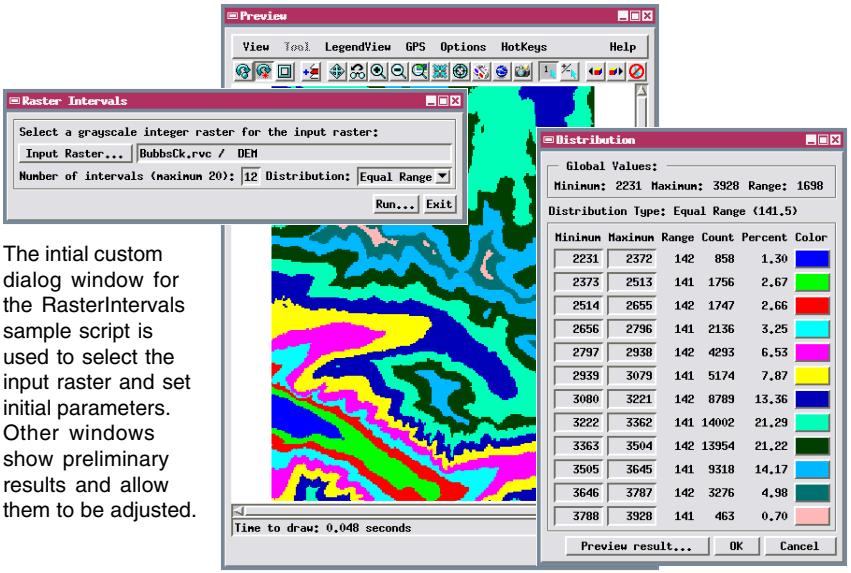
Custom Dialog Windows

Although popup dialog functions are easy to program and use, each function opens a separate transient dialog window. In a complex processing script requiring a number of inputs and parameter settings, using a separate popup dialog for each may make for a cumbersome user interface.

You can make a complex script easier to use by creating one or more custom dialog windows to provide a consistent and integrated user interface. A main script dialog can combine initial object selection and parameter settings, and other dialogs can be used for additional settings and presenting results. Custom dialog windows can include push buttons, menus, lists, and other components that you are familiar with in the TNTmips user interface. You can create dialogs with a drawing canvas to show graphic results or dialogs incorporating views of the resulting geospatial data. The Tutorial booklet *Building Dialogs in SML* provides a complete overview of procedures and techniques for creating and using your own custom dialog windows.



Custom dialog window for the sample script PanSharpComp.sml.



The intial custom dialog window for the RasterIntervals sample script is used to select the input raster and set initial parameters. Other windows show preliminary results and allow them to be adjusted.

Script Development and Editing

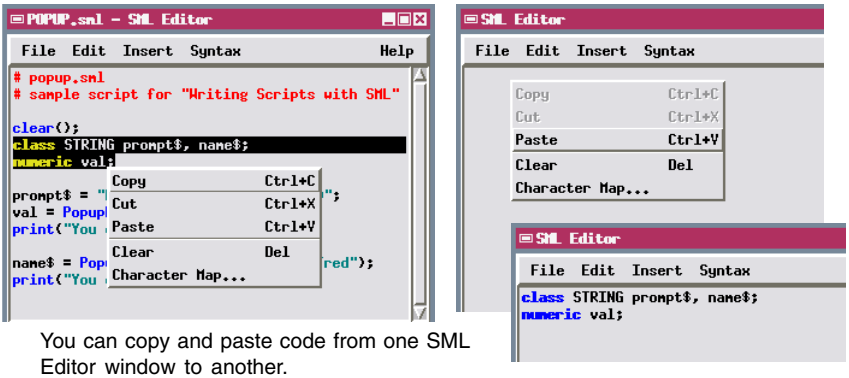
The Edit menu in the SML Editor window provides several options that let you navigate through a long, complex SML script. Use the Find option to search for a particular text string (such as a function name). The Find Again option repeats the previous search, allowing you to step through multiple instances of the search text. The Go to Line option moves the cursor to the designated line number in the script.

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, providing examples that process various types of spatial data and associated attributes. 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). 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.

STEPS

- ☑ keep the script from the previous exercise open
- ☑ choose Find from the Edit menu in the SML Editor
- ☑ in the Find window enter "Popup" as the text to search for and press [OK]
- ☑ choose Edit / Find Again
- ☑ choose Edit / Go to Line, enter 4 in the Prompt window, then press [OK]
- ☑ 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



You can copy and paste code from one SML Editor window to another.

Preprocessor Commands and Debugging

STEPS

- ☑ select File / Open and select DEBUG.SML from the SML directory
- ☑ 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 preprocessor directives can be inserted from the Keyword list in the Script Reference window:

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

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.

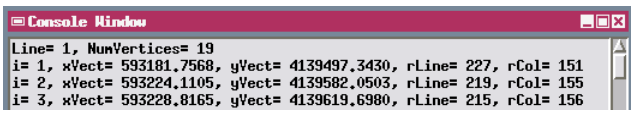
While you are developing a complex script you might want to have a "normal" mode of execution and a "debug" or “test” mode. In debug mode the current values of variables would be printed to the console at various points in the script to help you verify correct execution of intermediate steps and/or identify points of failure. You can set up the debugging mode using the $define directive to indicate your debug keyword

```
$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.



Note that preprocessor statements are *not* followed by a semicolon.

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.

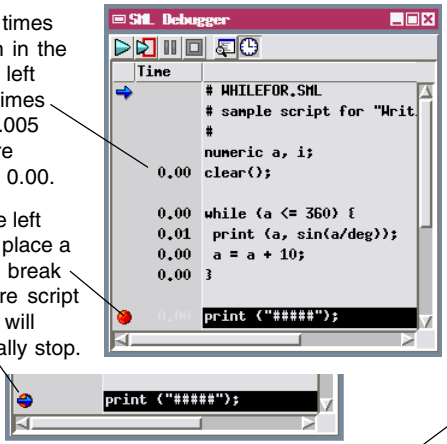
The SML Debugger window can also show the execution time (in seconds to hundredth-second accuracy) for each script step. For user-defined functions and procedures, cumulative times for one or more function calls are shown with the function definition, not where it is called in the script. You can use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

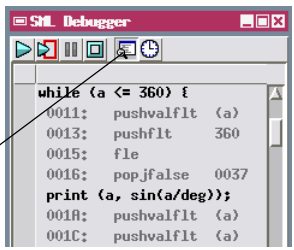
STEPS

- ☑ 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

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

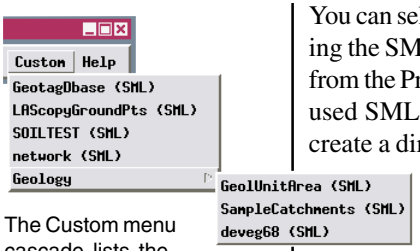
Click in the left column to place a temporary break point where script execution will automatically stop.





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

Toolbars and the SML Custom Menu



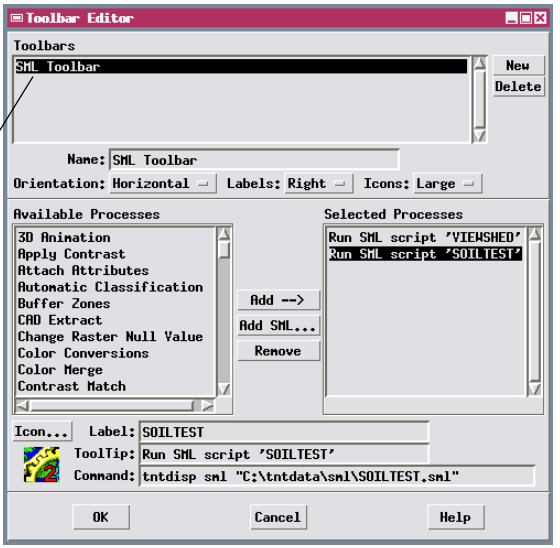
The Custom menu cascade lists the scripts in the CUSTOM directory in your TNT installation directory.

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.

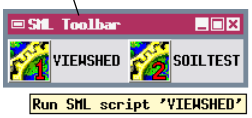
You can also assign SML scripts to icons on custom toolbars. Use the Toolbar Editor window to create or select a toolbar, set a horizontal or vertical orientation, and set up label positions. Then select one or more SML scripts and edit the Label and Tooltip text boxes as illustrated to establish the 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 Tools / User Toolbars / Edit from the TNTmips 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



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



Using Arrays and Matrices

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]
```

Arrays are commonly used in SML scripts to store lists of vector element numbers, line vertex coordinates, and the like. 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.

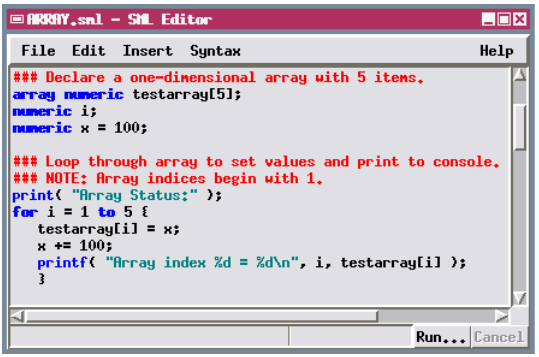
The MATRIX class implements a numeric matrix, which is always two-dimensional. You specify the matrix size as follows:

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

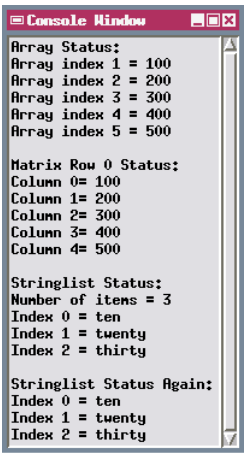
Matrix item values must be set and read using the SetMatrixItem() and GetMatrixItem() functions in the Matrix function group. Unlike arrays, indexing of matrix row-column positions begins with 0 rather than 1 (a three-column matrix would have columns 0, 1, and 2). The Matrix function group and Matrix class also include methods for inverting, transposing, and performing arithmetic operations on matrices.

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



- ☑ in the Script Reference window examine the documentation for the MATRIX class and Matrix function group



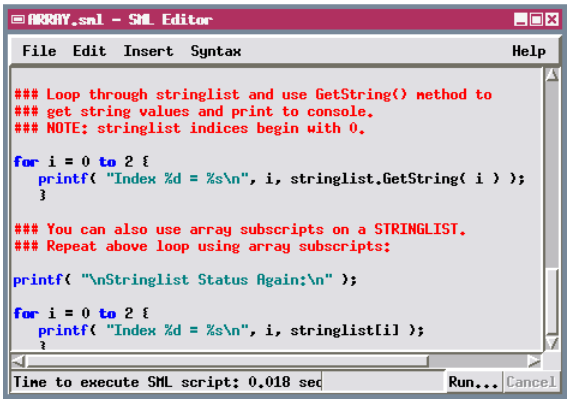
Stringlists and the DATETIME Class

STEPS

- ☑ examine the last part of the ARRAY.SML script from the previous exercise, which implements and uses a stringlist
- ☑ note the methods for reading values from the stringlist
- ☑ in the Script Reference window examine the documentation for the STRINGLIST class
- ☑ select File / Open and select DTSTRINGS.SML from the SML directory
- ☑ click [Run] to execute the script

The sample script introduced in the previous exercise also demonstrates the use of the STRINGLIST class, which is a structure to hold a list of text strings. Strings can be set or retrieved by their numeric position in the list using the SetString() and GetString() class methods; stringlist positions are indexed beginning at 0. You can also use array subscript notation to access strings in a stringlist.

The STRINGLIST class also includes methods to add strings to the front or end of the list, to remove a string, to remove duplicates, to swap two strings, to sort the list alphabetically, and to clear the list. The second example script for this exercise uses a stringlist to store the names of the months of the year to be retrieved using the number of the month.



This second script also demonstrates the use of the DATETIME class, a structure for storing and converting dates and times. This class includes a number of methods to set date and time using different standards and formats. The SetCurrent() method sets the date and time automatically from your computer's operating system, which in most cases is a local time. You can use the ConvertToUTC() method to convert a local time to Coordinated Universal Time, the international standard. You can set or retrieve dates in several numeric formats: as an integer designating the date as YYYYMMDD, or as a Julian date (the number of days since January 1, 4713 BC Greenwich noon in the Julian calendar, a system introduced by astronomers to provide a uniform system of dates).

- ☑ scroll through the script to see how the stringlist is created and used
- ☑ in the Script Reference window examine the documentation for the DATETIME class
- ☑ scroll through the script to see how the DATETIME class methods are used

Working with Text Strings

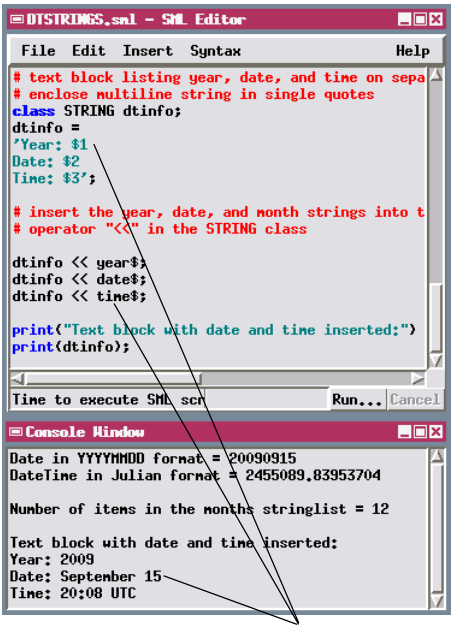
Text strings can be represented either by a string variable (the earliest representation in SML; see page 9) or by an instance of class STRING. Nearly all functions and class methods that take or return a text string now specify a class STRING, but you can pass them either a string variable or instance of class STRING. The sample script introduced in the previous exercise includes a number of examples of manipulating text strings.

STEPS

- ☑ in the Script Reference window, examine the functions in the String function group
- ☑ examine the methods and operators in the STRING class

The String function group includes a number of functions for processing text strings. The sprintf() function can be used to create precisely formatted text, to convert numeric values to strings, and to concatenate strings. The GetToken() function allows you to manually parse text using one or more delimiter characters.

The STRING class includes an even wider array of methods, including inserting, replacing, and removing a range of characters, extracting a substring using either beginning and ending positions or beginning position and length, and truncating the string. Character positions in a string are indexed beginning with 0.



The insertion operator can be used to sequentially insert the desired text strings, replacing numbered placeholders.

The STRING class also includes the + operator that allows you to manually concatenate strings, and logical operators (==, !=, <. >, <=, >=) for comparing the length and content of two strings. As shown in the illustration, you can also create text strings that include numbered placeholders ($1, $2, $3, etc.) and use the insertion operator (<<) to substitute the desired text. Each use of the insertion operator replaces the lowest-numbered remaining placeholder with the designated text.

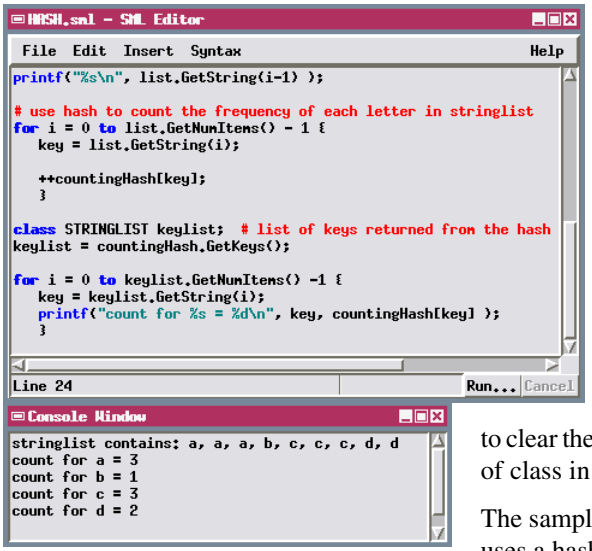
Using the HASH Class

STEPS

- ☑ select File / Open and select HASH.SML from the SML directory
- ☑ examine the script and its comments
- ☑ in the Script Reference window examine the documentation for the HASH class
- ☑ click [Run] to execute the script

The HASH class implements a structure somewhat similar to an array, except that a *key* is associated with each value in the structure. The key functions as a category label for each value. Any non-null string or numeric value can be used as a key. In addition, the values in a HASH are not restricted to numeric values, but can also be instances of any class. An instance of the HASH class thus is not declared directly. Instead a hash is constructed by declaring a variable or class instance followed by a pair of square brackets:

```
numeric  countingHash[];
class  COLOR  colors[];
class  RVC_OBJITEM  objItemList[];
```



Values are set or read from a hash like an array, using the key values as subscripts. The class also includes a method to get the list of keys from the hash as a stringlist, which you can use to iterate through the hash keys. It also includes methods to delete a particular key, to clear the hash, and to get the type of class in the hash.

The sample script for this exercise uses a hash to count the frequency of occurrence of the letters a, b, c, and d in a stringlist containing single-letter strings. The script gets each letter in the stringlist and increments the count in the hash for that key using the statement

```
++countingHash[key];
```

The key is automatically added to the hash if it does not yet exist.

A hash could also be used to sum attribute values for vector elements in different categories (such as polygon areas for different soil types in a vector soil map, for example).

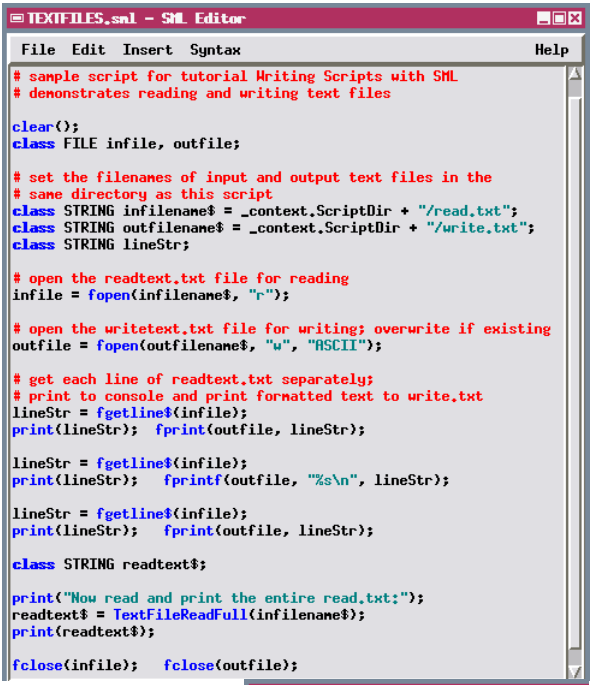
Working with Text Files

The files in the File function group allow an SML script to read text from and/or write text to a text file. For example, you might want a complex processing script to produce a log file that records all of the status information that you also direct to the SML Editor's Console window.

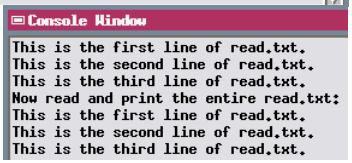
STEPS

- ☑ select File / Open and select TEXTFILES.SML from the SML directory
- ☑ examine the script and its comments
- ☑ in the Script Reference window examine the functions in the File function group
- ☑ click [Run] to execute the script

The fopen() function is used to open a text file for reading or writing and return an instance of class FILE. This function requires a filename string that includes the full directory path. If the desired text file is in the same directory as the SML script, you can use the CONTEXT class to construct the required directory path. A CONTEXT class instance called _context is automatically created by any SML script; the class member _context.ScriptDir records the path to the script directory.



The sample script reads the sample text file line by line using the fgetline$() function. There are also functions to read individual numbers, strings, and bytes from a file. For relatively short text files you can use the TextFileReadFull() function to read the entire file to a string variable.

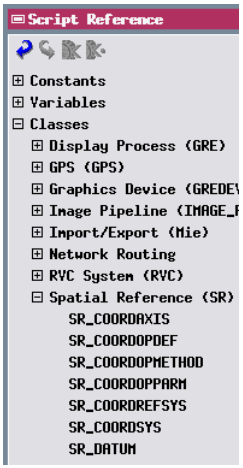


The File function group also includes fprint() and fprintf() functions to write strings to files; their syntax is similar to the print() and printf() functions.

Coordinate Reference Systems

STEPS

- ☑ select File / Open and select CRS.SML from the SML directory
- ☑ examine the script and its comments
- ☑ in the Script Reference window expand the Spatial Reference (SR) entry in the class list
- ☑ examine the documentation for the SR_COORDREFSYS class



MicroImages code numbers for coordinate reference systems and their components can be found on the Details tabbed panel of the Coordinate Reference System window (see the Coordinate Reference Systems tutorial). You can use the utility script CRSgetID.sml in the SML directory to interatively select a CRS and return its MicroImages and EPSG ID codes.

SML scripts that access map locations from a spatial object usually must deal with the object's Coordinate Reference System (CRS). SML classes associated with spatial referencing are found in the Spatial Reference (SR) class group.

Class SR_COORDREFSYS defines a complete coordinate reference system. The coordinate system, datum, and projection information in a CRS are represented, respectively, by the SR_COORDSYS, SR_DATUM, and SR_COORDOPDEF classes. There is a class member in SR_COORDREFSYS for each of these three classes.

A complete CRS (or any of its component classes) can be assigned using a text string with an identifier for that spatial referencing component. Some components have text identifiers that are self-describing, as in the case of the WGS84 / Geographic CRS set up in the first part of the sample script:

```
class SR_COORDREFSYS latlonCRS =
       "Geographic2D_WGS84_Deg";
```

The assignment can be done as part of the class declaration, as above, or after the class is declared using the Assign() method on each SR class. The reference documentation for the Assign() methods lists the long text identifiers available for each.

The identifier for a CRS or its components can also be a text string consisting of an ID number preceded by the codespace for this identifier, as in:

```
utmCRS.CoordSys = "EPSG:4400";
```

EPSG stands for European Petroleum Survey Group, which maintains a widely-used database of spatial reference components. The supported codespaces are listed in the documentation for the GetID() method on each of the SR classes. Internal MicroImages ID codes can be used without naming the codespace:

```
projectDef.Method = "1909";
```

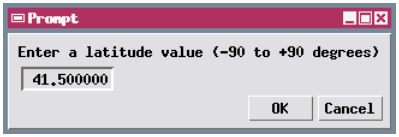
Coordinate Transformations

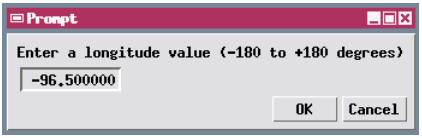
The sample script introduced in the previous exercise illustrates a common task in SML scripts: transforming point coordinates from one Coordinate Reference System to another. To simplify the example, no spatial objects are used; the script simply converts the entered latitude/longitude coordinates (referenced to the WGS84 datum) to UTM coordinates. The appropriate UTM zone is computed automatically from the entered location.

STEPS

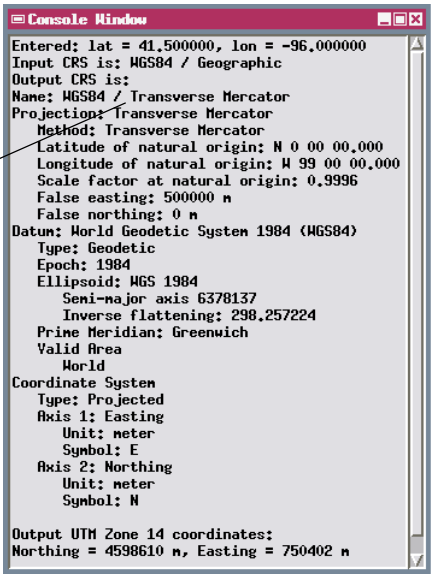
- ☑ in the Script Reference window examine the documentation for the TRANS2D_MAPGEN class
- ☑ click [Run] to execute the script
- ☑ enter latitude and longitude values in the Prompt windows as shown below

Class TRANS2D_MAPGEN is used to set up and perform coordinate transformations. The class includes methods to assign input and output coordinate reference systems. It also includes methods to perform forward or inverse transformations on 2D points, 3D points, and extents rectangles (class RECT). In this example the input CRS for the transformation is WGS84 / Geographic, while the output CRS is WGS84 / Transverse Mercator.





TRANS2D_MAPGEN and the spatial referencing (SR) classes in SML replace and extend the earlier classes TRANSPARM (now deprecated) and MAPPROJ.



Because some parameters of the Transverse Mercator projection used in the UTM system are specific to the UTM zone, this sample script sets all of the projection parameters individually. The resulting coordinate system and projection are equivalent to that of the corresponding UTM zone, but are not identified as “UTM zone 14N” in the CRS description printed to the console. In a script with a fixed output CRS using a UTM zone, you can set the defined projection (SR_COORDOPDEF) for the UTM zone at once using its identifier in any supported codespace.

Object and Map Coordinates

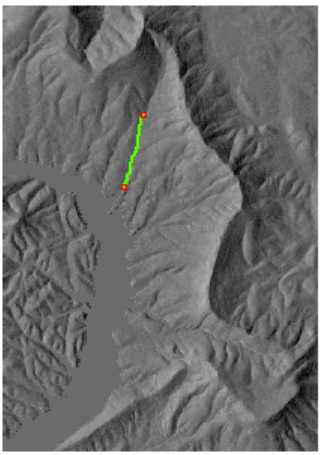
STEPS

- ☑ select File / Open and select OBJMAP.SML from the SML directory
- ☑ examine the script and its comments
- ☑ [Run] the script
- ☑ when prompted for an input raster, choose DEM from the VIEWSHED Project File in the SML directory
- ☑ when prompted for an input vector, choose VPATH from the VIEWSHED Project File
- ☑ examine the information printed to the console

Georeferenced spatial objects have two distinct types of coordinates for indicating positions within the object: object coordinates and map coordinates. *Object coordinates* are the values stored internally with the object that record the positions of its elements. For a vector or CAD object that was created by digitizing a drawing, for example, the object coordinates would be the arbitrary x and y coordinates of the digitizing tablet or computer screen. For a raster object, the object coordinates of a cell are its raster column (x) and raster line (y) number.

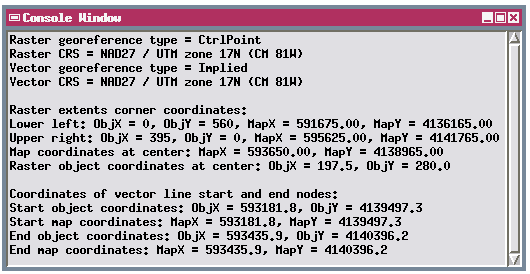
The *map coordinates* for an object are coordinates in the Coordinate Reference System assigned in the object's georeference. Map coordinates can be computed as needed in the TNT products from the corresponding object coordinates using parameters stored in the georeference subobject.

The RVC_GEOREFERENCE class includes a GetTransParm() method to get the transformation parameters between object and map coordinates as a TRANS2D_MAPGEN. You can then use the methods on this class to transform coordinates between these two systems. The sample script for this exercise reports object and map coordinates for the corners of a raster object and for the start and end nodes of a line in a vector object.



Raster object DEM and the line in vector object VPATH.

In many cases, the object coordinates of a geometric object (vector, CAD, shape, or region) are equivalent to its map coordinates. This is indicated by a georeference type of "Implied". However, an SML script should not assume that object and map coordinates of input geometric objects are identical.



Raster Objects

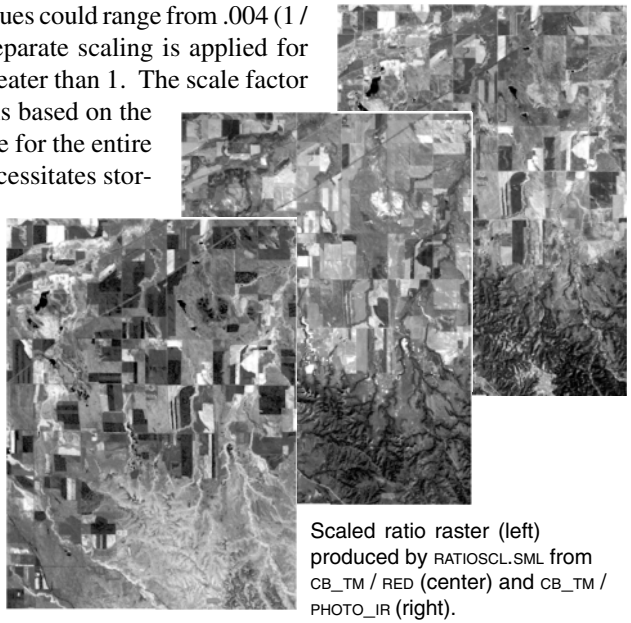
A full set of raster functions let your SML scripts read, create, and analyze raster objects. You can write mathematical expressions to compute values for a new raster object from one or more input rasters or use various higher-level SML functions to create new raster values.

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).

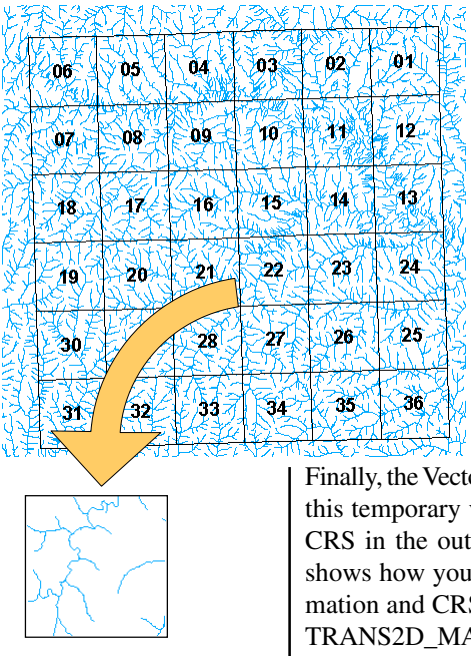
Vector Objects

STEPS

- ☑ select File / Open and select VECTOROPS.SML from the SML directory
- ☑ study the script structure and statement syntax
- ☑ run the script
- ☑ when prompted, select input vector objects SECTIONS and STREAMS from the VECTORDATA Project File in the SML directory
- ☑ create a new vector object for the output
- ☑ enter 22 in the popup dialog that prompts for the section number
- ☑ enter 2 in the popup dialog that prompts for stream order

SML scripts can also read, process, and create vector objects. A large set of functions for processing vector objects can be found in the Vector function group. The Vector Network and Vector Toolkit function groups provide more specialized vector functions.

An SML script is a great way to automate a series of processing steps that would require using separate processes if run interactively in TNTmips. The sample script for this exercise illustrates this concept. It starts with two input vector objects. One has polygons for numbered square-mile sections in a township, and the other has stream lines attributed with stream order data (numbers indicating the ranking of individual stream lines within the stream network hierarchy). The script creates an output vector object with stream lines for a selected section, with the selected minimum stream order, and reprojected to a different coordinate reference system.



The script uses three main functions to perform these operations. The VectorCopyElements() function is used to copy the section polygon (selected by query) to a temporary vector object. This temporary vector is used as the "operator" vector in the VectorExtract() function to extract and clip the stream lines (selected by query) for this section to a second temporary vector object. Finally, the VectorWarp() function is used to reproject this temporary vector to the WGS84 / Geographic CRS in the output vector object. The script also shows how you can obtain the georeference information and CRS for a spatial object and set up the TRANS2D_MAPGEN needed for the reprojection.

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);
```

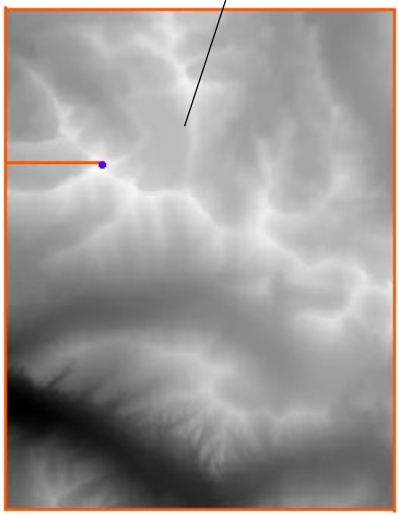
When you will be adding elements to a new output vector object, toolkit initialization can be done when the object is created. The second argument to the GetOutputVector() function is an optional flag string that can be used to set the topology level and to initialize the vector toolkit. For example, 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 coordinate reference system of the raster object and adds a vector line outlining the raster's rectangular extents. The location on this boundary line that is closest to the maximum cell point is then found, a line is added connecting these two locations. A point element is also added at the position of the maximum cell value. 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 raster DEM from the BUBBSCK Project File in the SML directory as input

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



This sample script also illustrates the use of the POLYLINE class, a structure for making and manipulating lines with any number of vertices.

Regions

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

The Region function group contains functions to open and save region objects

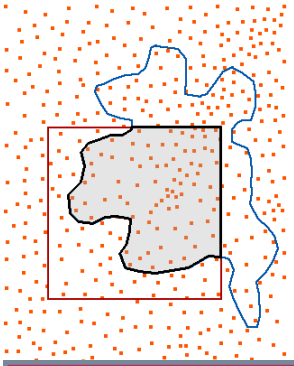
A *region* is a simple spatial object that represents the outline of an area of interest. In an SML script you can open an existing region object from a Project File or create temporary regions in many ways from other spatial objects (from the object's extents, from selected polygons, etc.). The script can then use these regions to operate on other spatial objects (test for inclusion, extract, etc.).

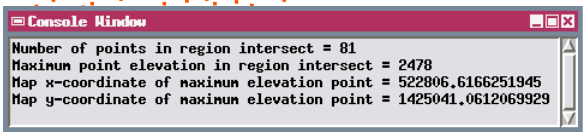
The REGION2D class includes methods for testing whether a point, line (class POLYLINE), rectangle (class RECT), or another region overlaps or is contained within the current region. There are also methods for obtaining the intersection, subtraction, or union of the current region with a polyline, rectangle, or another region (the current region is replaced by the result of the operation, so you don't have to create an additional region variable).

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 REGION2D class in SML represents a region that has been created in memory by some SML function or class method. Class REGION (derived from REGION2D) represents a region object in a Project File.

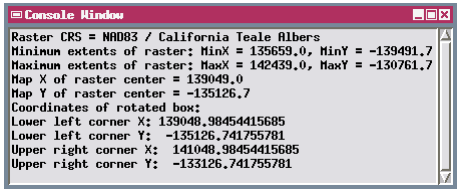
The sample script REGION.SML opens two region objects and finds their intersection. This new temporary region is then used to select points from a 3D vector object for examination and find the point with the maximum elevation (Z value).

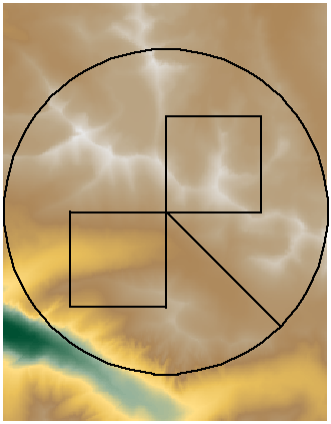
CAD and TIN Objects

SML includes classes and functions to 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.

STEPS

- ☑ select File / Open / *.SML File and open the script SML / CAD.SML
- ☑ examine and then run the script using raster object DEM from the BUBBSCKProject File in the SML directory

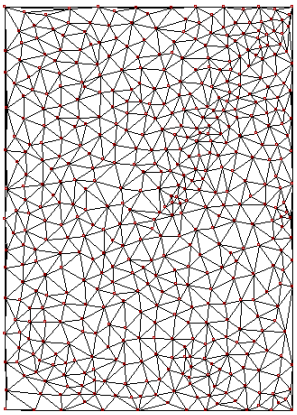


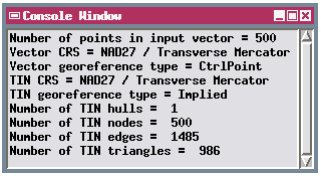


CAD object created by CAD.SML displayed over input raster DEM

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.

- ☑ 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



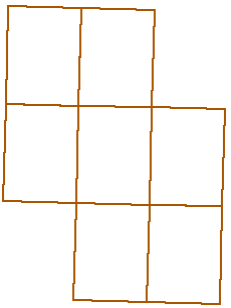


TIN object created from vector points using sample script TIN.SML

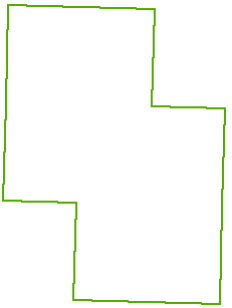
Shape Objects

STEPS

- ☑ select File / Open / *.SML File and open the script SHAPE.SML from the SML directory
- ☑ study the script structure and comments
- ☑ run the script using for input the shape file QUADS.SHP



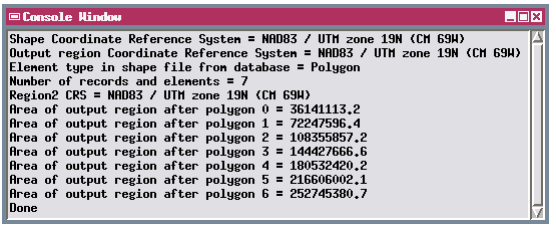
Input shape object with quadrangle boundary polygons



Region created by the SHAPE.SML script

The RVC_SHAPE class represents a shape object in SML. A shape object in a Project File links to an external shapefile or to spatial data stored in an external database (such as Oracle Spatial, PostGIS, MySQL Spatial, or an ESRI Geodatabase). The RVC_SHAPE class provides methods to read point, line, or polygon elements from a shape object for a particular element number. Shapefiles support multi-element constructs (more than one point, line, or polygon associated with the same record and element number), so the class methods for reading shape elements are structured to account for this possibility. The ReadPoints() method returns a polyline with vertices corresponding to the points for that element number; if there is only one point, it returns a degenerate polyline with only one vertex. The ReadLines() method returns an array of polylines (one for each line assigned to the element). The ReadPolygons() method returns a single region object created from the one or more polygons associated with the current polygon number.

The sample script for this exercise opens a shape object consisting of polygons (each corresponding to a separate element) outlining map quadrangles. It gets the region for each polygon and uses the UnionRegion() method of class REGION2D to add that region to an "accumulator" region. At the end, the "accumulator" region outlines the outer boundary of the set of quadrangle polygons, and this region is written to an output region object.



Reading Values from Database Tables

SML scripts can read and utilize attibute values from database tables associated with spatial objects. The simplest syntax for reading attributes is an extension of the TABLENAME.FIELDNAME construction used in queries. In an SML script that reads attributes attached to vector or TIN elements, a database field reference must also specify the spatial object, the element type (a separate database subobject is maintained for each type of element), and the element number. The database field reference has the following form:

object variable element type element number

```
num = Vect.poly[i].table.field;
string$ = Vect.point[i].table.field$;
```

Note that if the field being read is a string field, you must append the "$" character to the end of the field reference.

The vector object element type in this construct can be poly, line, point, or node. TIN object element type can be triangle, edge, or node.

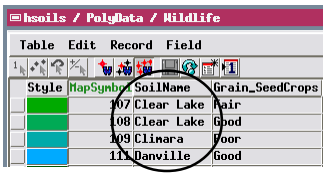
If there are multiple records attached to the element, the expressions above return the field value from the first attached record. To refer to the Nth attached record, use the following form:

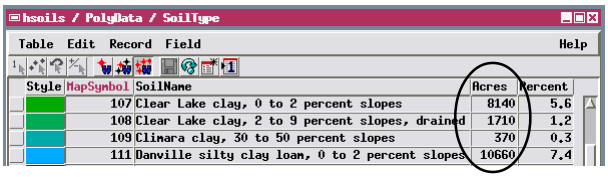
```
num = Vect.line[i].table[N].field;
```

The sample script for this exercise loops through all of the polygons in a soil map vector object and reads the Acres numeric field in the SoilType table and the SoilName string field in the Wildlife table:

```
for i = 1 to V.$Info.NumPolys
  {
  acres = V.poly[i].SoilType.Acres;
  type$ = V.poly[i].Wildlife.SoilName$;
```

These values are printed to the console window along with the element number.





STEPS

- ☑ open the sample script DB_READ1.SML from the SML directory
- ☑ study the script structure and comments
- ☑ run the script using object HSOILS from the HAYWSOIL Project File in the SF_DATA directory for input

DB_READ1.SML reads values from the ACRES field of the SOILTYPE table and the SOILNAME field of the WILDLIFE table.

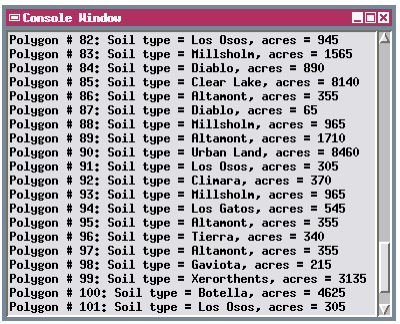
Using RVC DATABASE Classes

STEPS

- ☑ in the Script Reference window examine the documentation for the Database classes in the RVC SYSTEM class group
- ☑ open the sample script DB_READ2.SML from the SML directory
- ☑ study the script structure and comments
- ☑ run the script using object HSOILS from the HAYWSOIL Project File in the SF_DATA directory for input

The simple syntax presented on the previous page for reading database information cannot be used for CAD and Shape objects. Database-related classes in the RVC System group allow you to work with attributes of all types of spatial objects and are integrated with the RVC object classes. The RVC database classes also provide a highly structured and rigorous approach to reading and writing database information. The Database group within RVC System includes classes for databases attached to different types of spatial elements (such as RVC_DBASE_POINT and RVC_DBASE_CAD), the RVC_DBTABLE class that represents a database table, and the RVC_DBTABLE_RECORD class that provides a container for the contents of a record. Other classes in RVC System represent a record number and the spatial element a record is or can be attached to.

The script in this exercise replicates the same task as the DB_READ1 script in the previous exercise, but using the RVC System classes. For each soil polygon the script sets the element number for an instance of class RVC_ELEMENT and uses a method on class RVC_DBASE_POLYGON to get from the SoilType and Wildlife tables the record numbers of the records attached to this element. The results from each table are returned to an array of class RVC_RECORDNUM (since in general there may be more than one attached record). These record numbers are then used to read the contents of the corresponding record from each table to an instance of class RVC_DBTABLE_RECORD. A method on this class is then used to read (by field name) the values in the Acres field in the SoilType table and the SoilName field in the Wildlife table. These values are read to script variables and then printed to the console.



Creating an Element Database

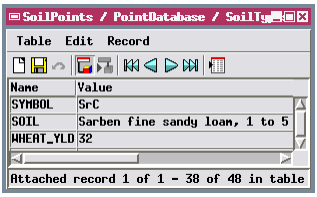
The two sample scripts in this exercise demonstrate two approaches to creating a new database and table in an output vector object, writing records with attributes read from the input object, and attaching these new records to the appropriate elements. DB_WRITE1 uses functions in the Database function group and corresponding classes. DB_WRITE2 uses the more recent RVC System classes and their methods to perform the same tasks.

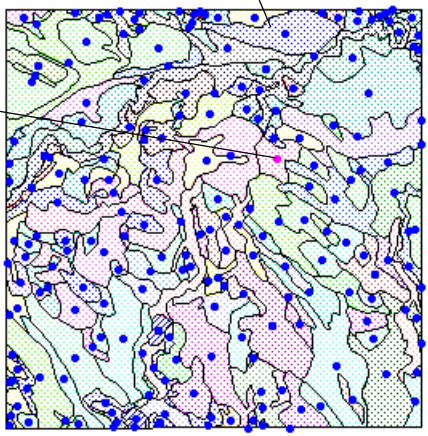
The scripts use the CBSOILS_LITE vector object as input and create an output vector object with a point at the centroid position of each soil polygon. These points are attributed with selected values from the input Class table (directly attached to the polygons) and other related tables. Because many records in the Class table are attached to more than one polygon, the scripts iterate through the Class table records to get the desired attributes from this table and from the related records in the other input tables and create a record in the output vector point database with these values. In addition, for each Class record the scripts get the list of attached soil polygons and iterate through these polygons to create the centroid point and attach the current new soil data record to each. Thus the output table contains no duplicate records.

STEPS

- ☑ open the sample script DB_WRITE1.SML from the SML directory
- ☑ study the script structure and comments
- ☑ run the script using object CBSOILS_LITE from the CB_SOILS Project File in the CB_DATA directory for input
- ☑ open the sample script DB_WRITE2.SML from the SML directory
- ☑ study the script structure and comments
- ☑ run the script using the same vector object for input

The CBSOILS_LITE vector object includes 212 soil polygons assigned to 48 soil classes. Some records in the Class table are thus attached to more than one soil polygon.





Sample scripts DB_WRITE1 and DB_WRITE2 create a new vector object with points located at the centroids of the polygons in the CBSOILS_LITE input vector. A record is created for each unique soil class in the input, populated with selected soil attributes, and attached to the relevant points.

Converting Objects

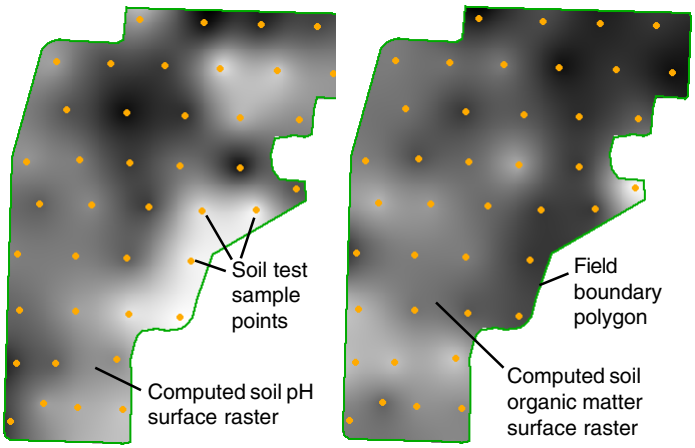
STEPS

- ☑ 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

```
RasterCompositeToBHS
RasterCompositeToCMYK
RasterCompositeToHIS
RasterCompositeToHSV
RasterCompositeToRGB
RasterRGBToComposite
RasterToCADBound
RasterToCADLine
RasterToTINIterative
RasterToTINIterative2
RasterToVectorBound
RasterToVectorContour
RasterToVectorContour2
RasterToVectorLine
TINToRaster
TINToVectorContour
VectorElementToRaster
VectorToBufferZone
VectorToBufferZoneExt
```

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.



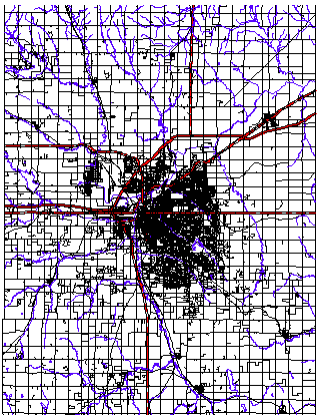
Sample Script: Extract Polygons

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.

STEPS

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

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.



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

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

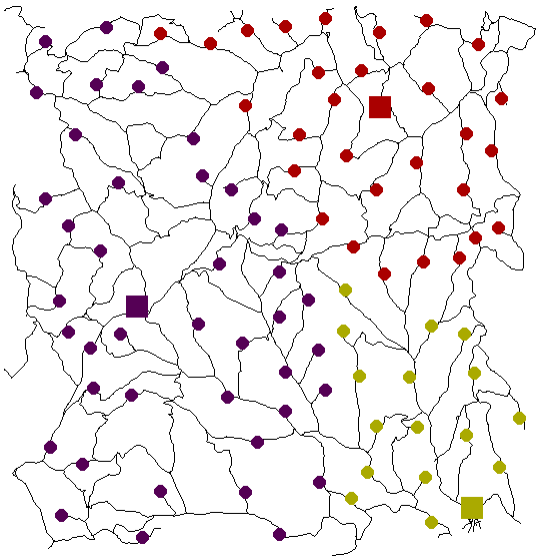
http://www.microimages.com/documentation/cplates/65smltiger.pdf

Sample Script: Network Routing

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

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.



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.

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. This script also uses the SetStatusMessage() function to post process status information to the status line at the bottom of the SML Editor.

More about the network script is available in an online document at

http://www.microimages.com/documentation/cplates/65smlnz.pdf

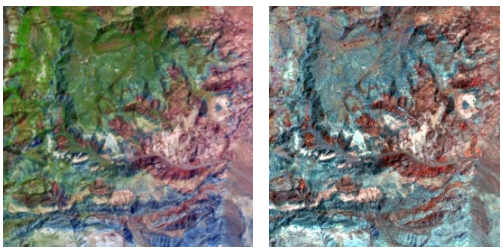
Sample Script: Devegetating Images

This exercise provides an example of custom image processing using SML. The sample script implements an automated procedure for suppressing the expression of vegetation in multispectral images for geological and soil-mapping applications. Using a method developed by NASA researchers, near-infrared and red image bands (in this example, Landsat bands 4 and 3, respectively) are used to estimate spatial variations in vegetation abundance. The script determines the statistical relationship between the brightness values in each selected image band and the vegetation index. For each band the cell values are then adjusted so that the average band value for each level of vegetation index is uniform across all index levels. This method works well in areas of open-canopy vegetation, such as arid and semi-arid terranes, where many image cells include both vegetation and bedrock or soil.

Since an image band may contain many cells with the same value, the script precomputes many values that depend solely on input cell value, then reads the computed values from the arrays as it iterates through the raster cells. In addition to adjusted image bands, the script produces a vegetation index raster and, for each devegetated band, a raster scatterplot of band versus vegetation index values and a CAD object containing a graph of the smoothed band average versus vegetation index value.

STEPS

- ☑ choose File / Open / *.SML File... and select DEVEG.SML from the SMLDLG directory
- ☑ run the script
- ☑ in the dialog window that opens, press [select NIR...], navigate into the INYOTM Project File in the SMLDLG directory, and select BAND_4
- ☑ press [Select RED...] and select BAND_3
- ☑ turn on the Devegetate NIR and Devegetate RED toggle buttons
- ☑ type "3" into the "Number of additional bands to devegetate" field and press <Enter>
- ☑ press [Select Bands...]
- ☑ select BAND_1, BAND_2, and BAND_5
- ☑ press [Set dark pixel values...]
- ☑ press [OK] in the Set Dark Pixel Values window that opens
- ☑ press [Select Output File...] and name a new Project File to contain the output objects
- ☑ press [Directory...] and select the directory in which TNTmips is installed
- ☑ press [OK] to start processing



RGB displays of Bands 5, 4, and 2 (respectively) for the input (left) and output (right) images. Vegetation (green in left image) has been effectively suppressed in the output image.

More about the network script is available in an online document at

http://www.microimages.com/documentation/cplates/68deveg.pdf

Sample Script: Processing LIDAR Points

STEPS

- ☑ choose File / Open / *.SML File SML / LAS_GROUND.SML
- ☑ study the script structure and comments
- ☑ run the script using LIDARCLS.LAS from the SML directory as input

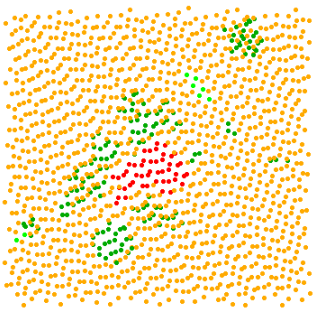
An SML script can read, process, and create LIDAR (LIght Detection and Ranging) point files in the standard LAS file format. TNTmips represents a linked LAS file as a shape object. Thus the RVC_SHAPE object class is used to work with LAS files in an SML script.

The sample script for this exercise is designed to process an LAS file whose LIDAR points have been classified into ground, vegetation, building, and other surface categories. The script makes a new LAS file using the MakeLAS() method on the RVC_SHAPE class, extracts the information for input points in the ground classification, and writes it to the new LAS file. The resulting ground points file would be suitable for generating a "bare-earth" digital elevation model (DEM) raster.

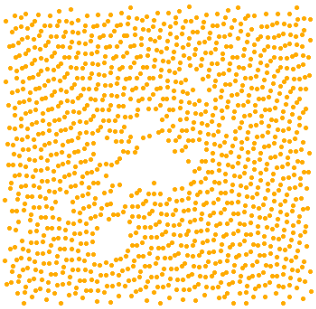
Extract of a natural-color orthoimage of the area covered by the sample LIDAR points. A building is surrounded by trees, a few lower shrubs, and open ground (grass and some nearly bare soil).

LAS files store 3D point coordinates, various LIDAR pulse parameters, and the point classification (if any) in a unique database record for each point. Thus all of the information necessary to define a LIDAR "point" can be copied to an output LAS file by copying the corresponding record from the main table (RVC_DBTABLE class instance) in the input LAS file to the main table in the output. There is no need to copy point "elements" when working with this special type of shape object.

- Ground
- Low Vegetation
- High Vegetation
- Building



Input LAS LIDAR file with classified points (indicated by point color).



Output LAS LIDAR file containing only ground points.

Using Status Dialogs

End-users of your scripts may run them without opening the SML Editor by choosing Script / Run from the TNTmips menu. A script run in this manner does not open a Console window to show the results of print statements used in the script to provide status information, and obviously the Editor's status line is also not available for that purpose. If you have designed a custom dialog for setting up and running your script, you can use it to show status messages (see the tutorial entitled *Building Dialogs in SML*). If you are not using a custom dialog, you can still provide status updates to the user by creating and using a predefined status dialog.

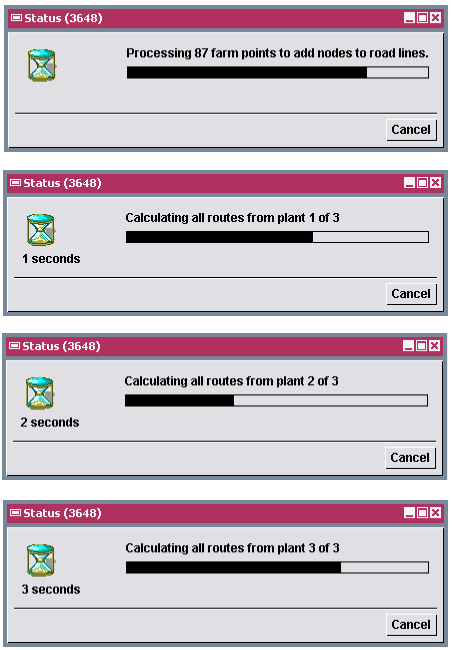
Two SML classes are involved in implementing a status dialog. You create an instance of the STATUSDIALOG class, then create a context (class STATUSCONTEXT) for it. The context provides methods for setting a message on the status dialog, initializing and incrementing a progress bar, and other dialog interactions.

The script for this exercise is a version of the vector network script introduced a few pages ago. This version adds a status dialog with progress bar that is re-initialized several times for the major script steps: 1) creating points in the road vector for the farms; 2) creating points in the road vector for the processing plants; and 3) iterating through the plants to perform a network analysis between the plant and all of the farms. In this third step the status message is changed and the status bar re-initialized for each plant loop, with the bar set to increment as each farm-plant route is computed.

STEPS

- ☑ choose File / Open / *.SML File SML / NETWORK2.SML
- ☑ note the use of the STATUSDIALOG and STATUSCONTEXT classes to implement a status dialog
- ☑ run the script using objects FARMS, PLANTS, and ROADS from the SML / NETWORK1 Project File

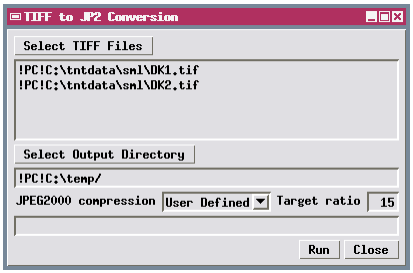
Note that on a fast modern computer this script may process this very small dataset too quickly for you to observe all of the updates made to the status dialog.



Batch File Conversion with SML

STEPS
- ☑ choose File / Open / *.SML File SML / TIFF_TO_JP2_MIE.SML
- ☑ study the script structure and comments
- ☑ run the script
- ☑ in the TIFF to JP2 Conversion dialog, press [Select TIFF Files] and choose DK1.TIF and DK2.TIF from the SML directory
- ☑ select an output directory
- ☑ set JPEG2000 compression options and press [Run]

You can use an SML script to automate repetitive tasks including importing or exporting tens or hundreds of data files with the same format. TNTmips supports the import and 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. (Note that files in GeoTIFF and many other widely-used geospatial file formats also can be used directly in the TNT products without requiring import.) Class members for each Mie class allow you to set process parameters such as raster size, compression, vector topology type, and others. These classes can be found in the Import/Export (MIE) class grouping.



Above, dialog created by the sample script. Below, the two sample images converted from GeoTIFF to GeoJP2.

The script for this exercise performs a batch conversion of any number of GeoTIFF files to GeoJP2 files with efficient JPEG2000 compression. You can choose from several JPEG2000 compression options and select an output directory. Each GeoJP2 file is automatically named for its input GeoTIFF file, which can contain a single composite image or any number of separate image bands. A temporary TNT Project File is created for each GeoTIFF file, and the MieGeoTIFF class is used to create one or more raster objects in this temporary file that link to the GeoTIFF file. These temporary rasters are then exported to a GeoJP2 file in the selected directory using the MieGeoJP2 class. Georeference information is automatically transferred to the output GeoJP2 files.

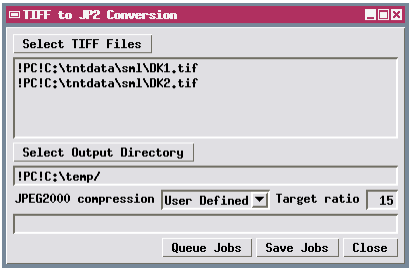
Running Scripts in Job Processing

You can also use SML scripts for batch-processing in the TNTmips Job Processing System. This allows you to run several scripts simultaneously, exploiting the processing power of your computer's multiple cores. To perform script-based batch job processing requires that you separate the interactive user input activities and processing activities into separate SML scripts. The interactive script provides the interface to choose one or more geospatial inputs and set global processing parameters. This script then makes a job file for each input. When each of these jobs is run, it calls the specified processing script and automatically passes the predefined variable values to it for execution. The process script must include variable declarations for all values passed from the job file. SML includes an MIJOB class with simple methods for automatically creating a properly-formatted job file, adding variable definitions and values to it, and specifying the process script to be run. Job files are automatically written to the TNT job directory you have set for your installation.

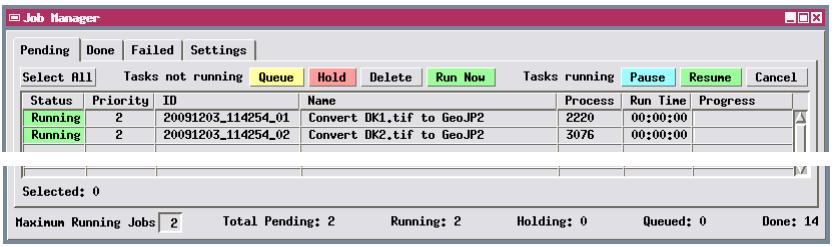
For this exercise, the GeoTIFF to GeoJP2 file conversion task introduced on the previous page has been implemented for job processing. The interactive script illustrates how to use the MIJOB class to create the required job files. The processing script uses preprocessor commands to define a test mode for testing the script by itself, outside of the job processing environment.

STEPS

- ☑ choose File / Open / *.SML File SML / TIFF_TO_JP2_MIE_FROMJOB.SML
- ☑ study the script structure and comments
- ☑ choose File / Open / *.SML File SML / TIFF_TO_JP2_MIE_MAKEJOBS.SML
- ☑ run the script
- ☑ in the TIFF to JP2 Conversion dialog, press [Select TIFF Files] and choose DK1.TIF and DK2.TIF from the SML directory
- ☑ select an output directory
- ☑ set JPEG2000 compression options and press [Queue Jobs]



Above, dialog created by the MAKEJOBS script. Below, the Job Manager running a job with the FROMJOB script for each of the input GeoTIFF files.



Pipeline Image Processing with SML

Pipeline Terminology

IMAGE: a raster object or file consisting of one band or color component, or a set of co-registered bands or color components.

STAGE: any pipeline element that represents or processes an image.

SOURCE: a stage that inputs an image.

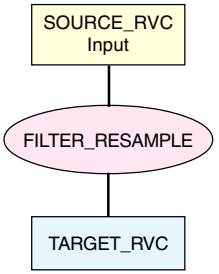
FILTER: a stage that applies some processing or transformation to the image.

TARGET: a stage that represents the final output image. Its properties are derived from the input stage it is connected to.

SAMPLE: the numeric value for a particular image row/column position and image component.

PIXEL: the set of SAMPLES for a particular image row/column position for multicomponent image.

Schematic Diagram of a Simple Image Pipeline



An image pipeline to resample and reproject an RVC raster image to an RVC raster image.

MicroImages has integrated *pipeline image-processing architecture* into SML, where it can be used in combination with the wide array of other functions and classes. A pipeline consists of a chain of processing elements arranged so that the output of each element (*stage*) is the input of the next. There are three types of stages (see more complete definitions in the box to the left): *source* (image input), *filter* (processing stage), and *target* (image output). Sources and targets can be raster objects in a MicroImages Project File or files in other formats supported for direct use in TNTgis. Filters are provided to perform a variety of operations such as resampling, mosaicking, applying spatial filters, cropping, applying a mask, and many others.

Each type of source, filter, and target is a separate SML class with its own predefined properties and methods (class functions). Pipeline connections are forged when a stage class is constructed in a pipeline script by specifying the previous stage that provides its input. A pipeline can have one or several sources, but only one target. Filters can be applied in series to one image or in parallel to multiple source images. Once the pipeline is constructed, a single method is called on the target stage to initiate processing and pull all of the image data through the pipeline.

Pipeline stages encapsulate their data, data properties, and operations, and they interact with each other in simple, defined ways. This modular, object-oriented design simplifies coding in SML and makes it easy to construct, modify, or extend a processing pipeline in a script. For example, georeference information is an inherent property of an image in an SML pipeline, so it is automatically pulled through the pipeline and assigned to the target. Likewise, pyramid tiers are automatically produced for target rasters in Project Files.

File Conversion via Image Pipeline

As a first example, we can look at a pipeline version of the TIFF to JPEG2000 format conversion script. The previous script processed each image in two stages using Mie classes: an import from TIFF format to an RVC-format raster in a temporary file, then an export to JPEG2000. The pipeline version used in this exercise is more efficient; pipeline source and target classes are provided for both formats, and the pipeline can convert each image directly without the intermediate conversions to/from RVC and without needing to manage any temporary files.

This format conversion is the simplest possible form of image pipeline: it consists only of a source and a target, because no other processing (filter stage) needs to be interposed between them in this case. The SOURCE_TIFF class reads the TIFF file and the conversion to JPEG2000 format is handled by the TARGET_J2K class. However, any number of filter stages could be added if needed to provide additional processing (such as resampling or applying contrast enhancement to the image).

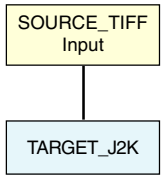
Source and target classes for external image formats are constructed using an instance of the FILEPATH class to indicate the location of the source or target file. Target classes for TIFF and JPEG2000 images have auxiliary settings classes with members that allow you to specify options for these more complex image formats.

Pipelines are set up sequentially starting with the source(s), since each subsequent stage must specify its input stage. Each target class has an Initialize() method that must be called to initialize the pipeline and check for valid connections between all of the stages. An error value (negative number) is returned if there are not, and the script should check for this condition. If the returned value is not negative, the script can call the Process() method on the target class to execute the pipeline processing.

STEPS

- ☑ choose File / Open / *.SML File SML / TIFF_TO_JP2_PIPELINE.SML
- ☑ study the script structure and comments
- ☑ run the script
- ☑ in the TIFF to JP2 Conversion dialog, press [Select TIFF Files] and choose DK1.TIF and DK2.TIF from the SML directory
- ☑ select an output directory
- ☑ set JPEG2000 compression options and press [Run]

Pipeline Schematic



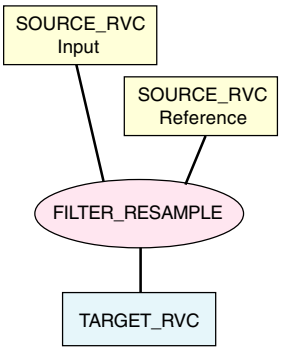
Pipeline processing in SML is described in more detail in the following Geospatial Scripting Technical Guides: *Pipeline Image Processing*, *Pipeline Programming Basics*, *Pipeline Structures for Multiple Inputs*, and *Using Regions in a Pipeline*.

Resample to Match Reference via Pipeline

STEPS

- ☑ choose File / Open / *.SML File SML / PIPELINERESAMPLETOMATCH.SML
- ☑ study the script structure and comments
- ☑ in the Script Reference window, navigate to the Classes / Image Pipeline / Filter list and highlight the entry for the FILTER_RESAMPLE class
- ☑ note the different forms of the constructor for this class

Pipeline Schematic



The script in this example is a slightly more complicated image pipeline that resamples the source image to match the extents, cell size, and coordinate reference system of a reference image. Both sources and the target in this example are RVC raster objects, but the same pipeline structure can be used with different types of sources and targets.

An RVC source or target is constructed with an instance of the class RVC_OBJITEM to specify the raster object to use or create. In this script the function DlgGetObject is used to open an Object Selection Dialog so the user can select the source image to resample, the reference image, and the location of the target image. This function encodes the raster filepath and object path as an RVC_OBJITEM class instance that can then be used in the constructor for a source or target.

The IMAGE_PIPELINE_FILTER_RESAMPLE class has several constructors with different sets of parameters. The choice of constructor determines whether the filter resamples 1) from object to map coordinates and a specified image size, 2) to a reference image, 3) to a specified coordinate reference system and cell size, or 4) to a specified coordinate reference system and image size. In this example the constructor that specifies a reference image is used.

A script can include more than one pipeline structure to perform different operations, with the target of one pipeline being used as the source for the next. The filepath or object path of a target are not valid until the pipeline is processed. The TARGET_RVC class has a GetObjItem() method (and the other TARGET classes a GetFilepath()method) that must be called after the pipeline is processed to get the valid path information (RVC_OBJITEM or FILEPATH) to pass along to the source of the next pipeline.

Script Builder

The Script Builder in TNTmips Pro provides a graphical design environment for creating standalone processing scripts. The Builder window provides a large design canvas and a sidebar list from which you can choose classes and functions to place in the canvas. The design canvas shows these script components as boxes that can be moved as needed and minimized or expanded to hide or show the parameters of the function or class. Connecting the components (as shown by the gray lines in the canvas) establishes the sequence of execution and automatically generates SML code that can be viewed and run using the Script tabbed panel. The Script Builder is best suited for creating scripts that use the pipeline processing classes. Script designs can be saved to and reloaded from a design file with the .smlb file extension.

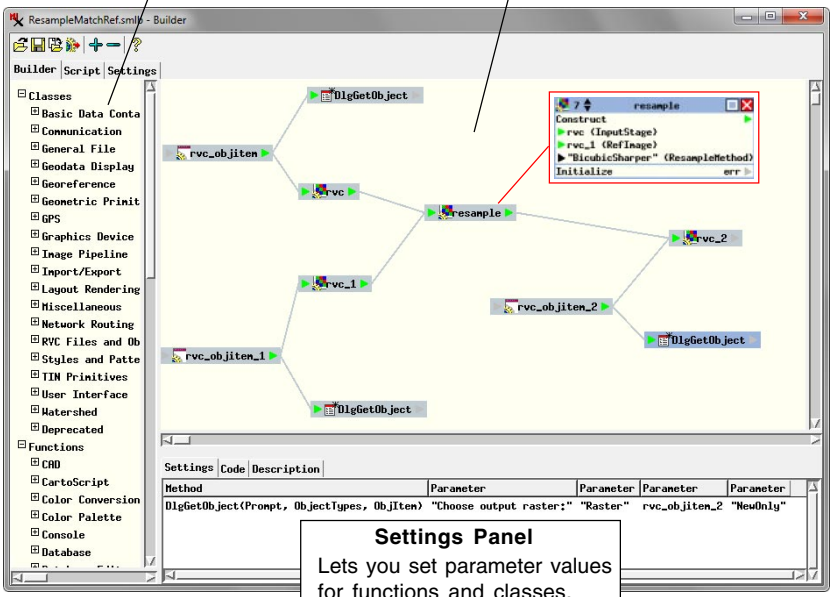
STEPS

- ☑ choose Script / Builder from the TNTmips menu
- ☑ in the Script Builder window, press the Open icon button
- ☑ choose SML / RESAMPLEMATCHREF.SMLB
- ☑ pause the cursor over a box in the design canvas to see its details
- ☑ open the Script tabbed panel to see the script generated from this design

For more information see the TechGuides entitled *Use Graphical Interface to Design Scripts* and *Operating the Builder.*

Class & Function List
Select components from list to place in the canvas.

Design Canvas
Functions and classes shown as boxes that you can position and connect to establish processing sequence.



Settings Panel
Lets you set parameter values for functions and classes.

Automated Processing with TNTscript

TNTscript is a professional product designed to enable automated production processing of geospatial data using SML scripts. TNTscript enables SML scripts to be executed on computers that do not have any other professional TNTgis products installed. If you develop SML processing scripts using TNTmips Pro or TNTedit Pro, TNTscript allows you to execute these scripts on additional computers, at remote sites, or using cloud-computing resources. TNTscript can also be used separately from the other TNTgis professional products to write, edit, and execute SML programs.

Runtime TNTscript is a Windows or Mac executable program that provides all of the noninteractive SML processing functionality found in TNTmips Pro. TNTscript executes SML scripts without user interaction during processing. It does not have a user interface, but instead is launched from the command line manually or by another program. Processing scripts written for use with TNTscript are therefore structured in a similar manner as those used in TNTmips job processing. All data-specific variables must be passed to the SML script by TNTscript at the start of execution. The script to be used and the required variable values can be specified on the command line or in an XML-formatted text file (.smlx).

TNTscript can be used to integrate custom SML processing into a production workflow involving other scripts and software products. A custom script controlling the work–flow can call TNTscript as needed and specify the name of an SML script and the required script parameters. The data produced by this processing script can then be passed along to the next step in the overall work–flow. Network licenses for TNTscript are available to allow TNTscript processing to be distributed over multiple networked computers. Licenses are also available for cloud-computing resources. TNTscript can therefore be used to integrate the capabilities of SML into geospatial processing workflows in an enterprise network to automatically create geospatial products as new data arrive, or in a web application that processes data by user request.

A TNTscript license includes the runtime TNTscript executable (with its own installer) and a professional license for TNTview (optional installation using the TNTgis installer). TNTview provides access to the SML Script Editor for writing, editing, and testing SML scripts for use with TNTscript. When you set the target product to TNTscript, the built-in Script Reference documentation indicates which functions and classes are available (everything except those related to interactive components such as pop-up dialogs, View windows, etc.). Fully interactive scripts can also be written and executed from TNTview under the TNTscript license.

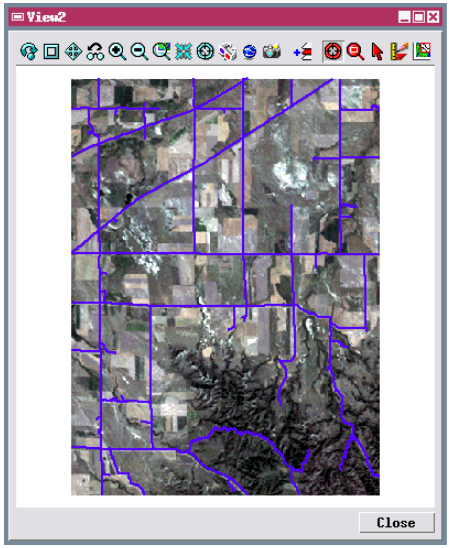
Creating and Opening a View Window

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.

Sample script VIEW2.SML shows the basic steps required to open a view window of a group and display several geospatial objects. A more detailed explanation of a similar script can be found in the tutorial booklet *Building Dialogs in SML*. That tutorial provides an introduction to creating all types of custom dialog windows for your SML scripts. It includes several additional examples of scripts that create dialogs incorporating geodata views.

STEPS

- ☑ in the SML Editor window select File / Open / *.SML File and select SML / VIEW2.SML
- ☑ run the script
- ☑ choose as input raster _8_BIT from the CB_COMP Project File in the CB_DATA sample data directory
- ☑ choose as input vector ROADS from the CB_DLG Project File in the CB_DATA sample data directory
- ☑ press [Close] to close the view window



Spatial objects added to a view become *layers* in a *group*, and the group may be contained in a *layout*. SML includes individual classes corresponding to particular layer types (such as raster layers, vector layers, and so on), and for groups, layouts, and the view itself. These classes all begin with "GRE_" and are found in the Display Process (GRE) class grouping in the Script Reference. Functions for creating and manipulating these display components are found in several Geodata Display function groups.

The script in the next exercise displays several data layers and provides a graphical point tool for obtaining coordinate information from the View.

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 the tutorial *Introduction to Geospatial Scripting*.

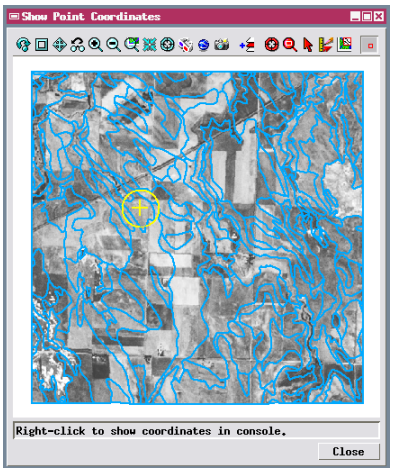
Coordinate Systems in Views

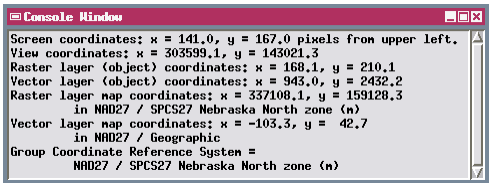
STEPS

- ☑ select File / Open / *.SML File and choose SML / PTCOORD2.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

Previous exercises have discussed SML class methods 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 PTCOORD2.SML will help you explore these coordinate systems and illustrates the resources available to convert between them. The script displays a preset raster and vector object and provides a graphic point 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.

A graphic tool in a view returns positions in the pixel coordinates of the drawing area of the view (measured from the upper left corner). These *screen coordinates* are used if you want to use SML drawing functions to draw additional features in the view. The view also has *view coordinates*, which for a single group view are the group map coordinates. The group coordinate reference system is determined initially by the georeference of the first layer added to the group, but can be modified by a script using a method on the GRE_GROUP class. 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*, which are determined by the georeference used by the object in the layer. The GRE_VIEW class includes methods for obtaining transformations between view coordinates and any of these other coordinate systems.





Working with Geodata in a View

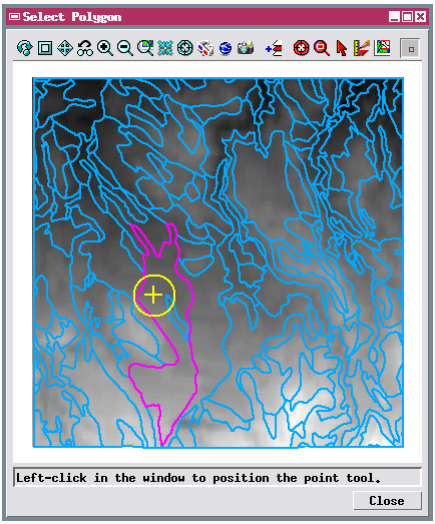
Graphic tools added to a view allow the script user to interact with the geodata layers in the view and obtain information from them that the script can then use to perform computations. The sample script for this exercise automatically displays a digital elevation model (DEM) raster object and overlying soil polygons. It also provides a point tool that is used to select a soil polygon and compute the average elevation from the corresponding area of the DEM.

STEPS

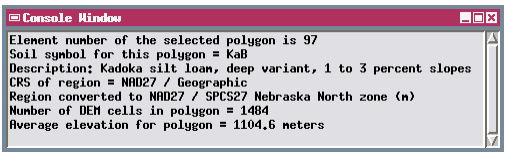
- ☑ select File / Open / *.SML File and choose SML / VIEWGD.SML
- ☑ run the script
- ☑ left-click in the window to place the point tool
- ☑ right-click to select and highlight a polygon and see data for that area printed to the console
- ☑ try various point locations to see how the polygon attributes vary
- ☑ Close the Select Polygon window when you are finished

The point tool is set as the default tool for this view. The tool is placed by left-clicking in the view, and right-clicking triggers the associated user-defined function. The script transforms the point tool position from screen to view coordinates and then from view to layer coordinates for the soil layer. The resulting object coordinates are passed to the FindClosestPoly() function, which returns the element number of the polygon enclosing the point. This element is highlighted, and some soil attributes are read and printed to the console. The polygon is converted to a region in the soil vector map coordinates, and this region is reprojected to the coordinate reference system of the DEM. A special loop construction (introduced in a previous exercise on Regions) is used to loop through just the DEM cells lying within this reprojected region:

```
for each DEM[lin,col] in reg {
  ...
  }
```



The DEM cells within the region are counted and their values are summed, allowing the average elevation for the region to be computed.



Movie Generation Scripts

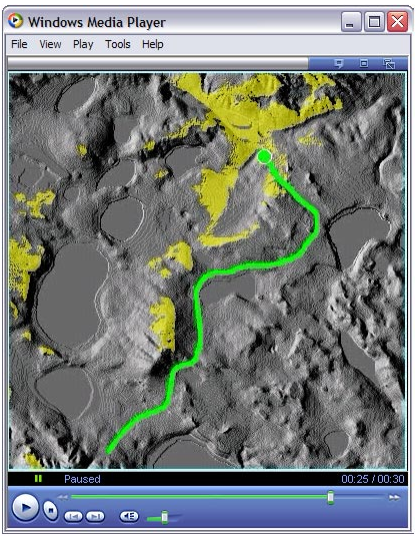
STEPS

- ☑ choose File / Open / *.SML File and select SML / VSHEDMOV.SML
- ☑ study the script structure and comments
- ☑ run the script
- ☑ for the input elevation raster choose PONDS from the PONDVIEW Project File in the SML sample data directory
- ☑ for the input vector choose VECPATH from the PONDVIEW Project File
- ☑ for the input style object choose STYLES from the PONDVIEW Project File

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

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

Sequential changes in the View window can be achieved in several ways. The script could add and remove a series of pre-prepared layers to and from the view. It could also modify the display parameters for a single continuing layer. For vector objects, this could involve basing the element styles on a sequence of varying attribute values (such as population in different years). The final method is exemplified by the VSHEDMOV3 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.



Modifying and Rendering Layouts

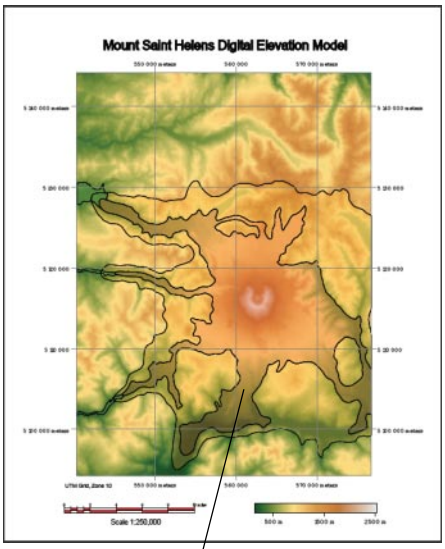
Layouts in the TNTmips Display process are used to assemble multiple geospatial data layers for presentation purposes. Display layouts form the basis for electronic atlases and can be rendered to KML files for viewing in Google Earth. Hardcopy layouts can include cartographic elements such as text annotation, scale bars, and legends for printing or rendering to PDF files for electronic distribution.

STEPS

- ☑ choose File / Open / *.SML File and select SML / LAYOUT.SML
- ☑ study the script structure and comments
- ☑ run the script, choosing COLOR / COLOR.RVC as the input Project File

The GRE_LAYOUT class in SML represents a layout object. Using this and associated classes and functions you can write SML scripts that read and modify an existing layout, create a new layout, or write a layout to a Project File. Your script can add or remove spatial layers from any group in the layout, set up styling and DataTips for new spatial layers, and modify text annotations in the layout. PDF and KML classes are provided in SML to allow you to render a completed layout to the respective file type.

The modified layout rendered to a PDF file and viewed in Adobe Reader.



The HazZones vector layer added to the layout by the script.

The sample script for this exercise reads a hardcopy layout from the TNT sample data, adds an existing vector object to the spatial group, and renders the modified layout to a PDF file. A more complex application might import geospatial data from one or more sources, process the data (such as extracting a desired area), and add the processed data to the layout before rendering. The PDF class also allows you to write additional pages to an existing PDF file, allow a script to generate multipage report files.

The sample *canvquakes* script provides a more complex example of modifying and rendering a layout. This script is run hourly at MicroImages by the TNTmips Pro Job Processing system to download and process earthquake data, add it to a layout with reference data, and render the layout to a KML file. This script, sample data, and explanatory Technical Guides are available for free download at www.microimages.com/downloads/smlscripts.htm.

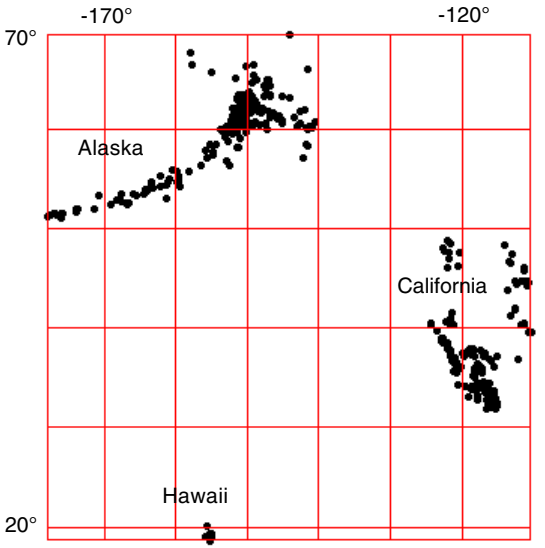
Accessing Web Data

STEPS

- ☑ choose File / Open / *.SML File and choose SML / DOWNLOAD.SML
- ☑ study the script structure and comments
- ☑ run the script
- ☑ when prompted, enter 17 for the minimum latitude, 70 for the maximum latitude, -178 for the minimum longitude, and -108 for the maximum longitude

The Internet is now an important source of free, publicly available geospatial data, some of which is updated frequently to represent current conditions. You can download files from the internet in an SML script using methods in the HTTP_CLIENT class. The Connect() method of this class takes a string parameter containing the identity of the remote host (either as an IP address or the host or website URL). It is a good idea to set a timeout time (in seconds) for the class using the SetTimeOut() method in case the script is unable to make the connection, otherwise it will keep trying to connect indefinitely and the script will never complete. Use the DownloadFile() class method to download the remote file. This method takes a string with the full URL to the remote file and a string with the local path and filename where you want the downloaded file to be written.

The sample script in this exercise downloads a text file containing global earthquake epicenter locations and attributes from a US Geological Survey website. It imports the global points to a temporary vector object and copies points to the output vector that are within the range of latitude and longitude you specify. The limits used in the instructions for this exercise extract points for the northeast Pacific region, including southern Alaska, California, and Hawaii. This script is a much-simplified version of the *canvquakes.sml* sample script referenced on the previous page.



Downloaded, imported, and extracted earthquake epicenter points for a 7-day period for the northeast Pacific Ocean region, displayed with a map grid.

Launching Other Programs

A geospatial process script can export data to various external file formats and launch another program to present or further process that data. The System function group provides several functions that you can use launch another program.

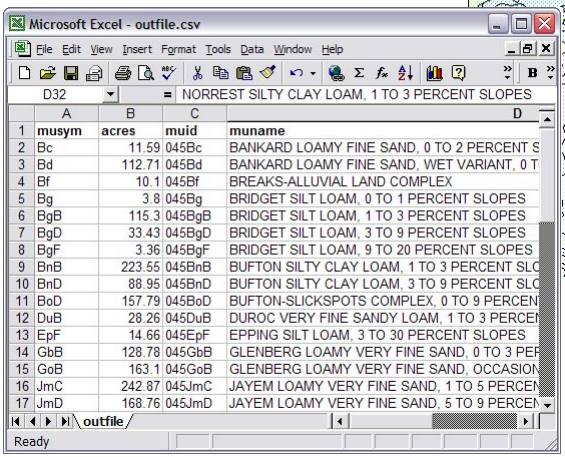
The run() function can be used to run a program but has no provision for passing command-line parameters or passing the data you have exported. The ExecuteProcess() function takes a process string that can include command-line parameters for the program you are calling. For example, the code below is used to start a PHP script:

```
start$ = "php c:/wamp/www/start.php " +
    taskID$;
ExecuteProcess(start$);
```

The RunAssociatedApplication() function takes a filename and uses your operating system resources to open the file in the program that your system has registered for that file type. The sample script for this exercise writes attribute data to a CSV text file and launches the spreadsheet program associated with that file type (for example, MicroSoft Excel).

STEPS

- ☑ choose File / Open / *.SML File and choose SML / SOILDATA.SML
- ☑ study the script structure and comments
- ☑ run the script
- ☑ for the input vector object choose CBSOILS_LITE from the CB_SOILS Project File in the CB_DATA sample data directory



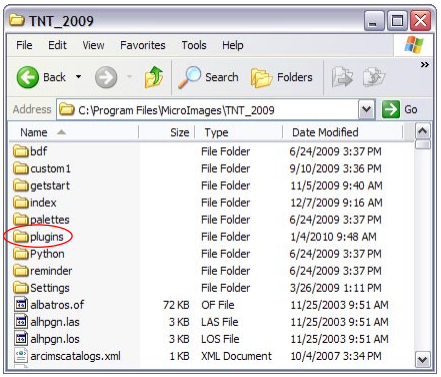
	A	B	C	D
1	musym	acres	muid	muname
2	Bc	11.59	045Bc	BANKARD LOAMY FINE SAND, 0 TO 2 PERCENT S
3	Bd	112.71	045Bd	BANKARD LOAMY FINE SAND, WET VARIANT, 0 T
4	Bf	10.1	045Bf	BREAKS-ALLUVIAL LAND COMPLEX
5	Bg	3.8	045Bg	BRIDGET SILT LOAM, 0 TO 1 PERCENT SLOPES
6	BgB	115.3	045BgB	BRIDGET SILT LOAM, 1 TO 3 PERCENT SLOPES
7	BgD	33.43	045BgD	BRIDGET SILT LOAM, 3 TO 9 PERCENT SLOPES
8	BgF	3.36	045BgF	BRIDGET SILT LOAM, 9 TO 20 PERCENT SLOPES
9	BnB	223.55	045BnB	BUFTON SILTY CLAY LOAM, 1 TO 3 PERCENT SLO
10	BnD	88.95	045BnD	BUFTON SILTY CLAY LOAM, 3 TO 9 PERCENT SLO
11	BoD	157.79	045BoD	BUFTON-SLICKSPOTS COMPLEX, 0 TO 9 PERCEN
12	DuB	28.26	045DuB	DUROC VERY FINE SANDY LOAM, 1 TO 3 PERCEN
13	EpF	14.66	045EpF	EPPING SILT LOAM, 3 TO 30 PERCENT SLOPES
14	GbB	128.78	045GbB	GLENBERG LOAMY VERY FINE SAND, 0 TO 3 PER
15	GoB	163.1	045GoB	GLENBERG LOAMY VERY FINE SAND, OCCASION
16	JmC	242.87	045JmC	JAYEM LOAMY VERY FINE SAND, 1 TO 5 PERCEN
17	JmD	168.76	045JmD	JAYEM LOAMY VERY FINE SAND, 5 TO 9 PERCEN

Soil attributes compiled from the CBSOILS_LITE vector object (above) output to a CSV file that is opened in a spreadsheet program (right).

Extending SML

Experienced programmers can add compiled functions to SML using the TNTsdk and/or C++. Create a Dynamic Linked Library (DLL) containing your functions and place them in a directory called "plugins" in the directory containing the TNTmips executables and DLLs. A sample program called smlplug.c provided with the TNTsdk shows how to include your compiled functions in SML. Your custom functions are accessable from the function list in the Script Reference window in their own function group.

TNTsdk is a free software development kit that you can use to create custom processes for TNTmips, TNTedit, or TNTview. TNTsdk provides programming libraries of several thousand C functions and hundreds of C++ classes. Your custom processes can invoke these MicroImages library components to present the user interface, operate on geodata, and present process results. Use of the TNTsdk requires a TNTmips Pro license. More information is available in the tutorial entitled *Introduction to Using TNTsdk*.



Create a directory named "plugins" in your TNTmips product directory and place the DLL with your custom functions there.

In TNTgis installations on Windows platforms, SML scripts can also launch and communicate with ActiveX component programs created in Visual Basic, C#, or C++. An SML script can directly access component class structures (data structures and methods), open a dialog window defined in the ActiveX component, and exchange data with the component class. Any information processed by the component program can be transmitted to other application programs or sent back to SML to be processed and/or written to the geospatial data in your Project Files. Experienced Windows programmers can therefore use the dialog design tools in Visual Basic to set up the user interface for an SML script rather than using the built-in user-interface components in SML.

To implement an ActiveX compent written in Visual Basic for use in TNTmips:

- design a Visual Basic form (dialog) and supporting properties and methods
- use Visual Studio to build an installation package to install and register the ActiveX component program
- write an SML script that uses the $import preprocessor command to "import" the ActiveX component class.

Communicate with ActiveX Programs

The sample script for this exercise illustrates the use of an ActiveX component written in Visual Basic to provide the user interface for an SML script. The script statement

```
$import  VB_PanSharp.VBForm
```

"imports" the VBForm class from the ActiveX program VB_PanSharp. The associated dialog is a *modal* dialog, meaning that SML script execution is suspended while the ActiveX dialog is open. The SML script can pass data to the imported component class before the dialog opens. and, as in this example, retrieve data (the dialog settings selected by the user, stored in the VBForm class data structure) from the imported class after the modal dialog has been closed.

With a *modeless* ActiveX dialog, the SML script can continue to execute while the ActiveX dialog is open, which requires on-going communication between the ActiveX class and SML. When you create the Active X component class, you can define *events* associated with the dialog controls, such as pressing a particular pushbutton. Each event should be provided with an associated class method that can be used in the SML script to register the name of a function elsewhere in the script that will be called and executed in response to that dialog event.

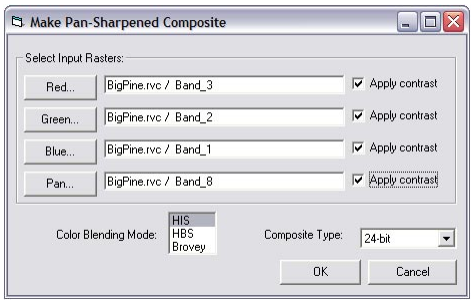
An ActiveX component created in Visual Basic can be compiled as a simple executable file or as a DLL. The latter method allows multiple instances of the component to run simultaneously. A modal ActiveX dialog can be activated from either form of component. An ActiveX component using a modeless dialog must be compiled as a simple executable file.

STEPS

- ☑ choose File / Open / *.SML File and choose SML / VB_PANSHARP.SML
- ☑ study the script structure and comments

To run this script on a Windows computer:

- ☑ download VB_PANSHARP.ZIP from www.microimages.com/downloads/smlscripts.htm
- ☑ unzip the file and run the Setup program in the Package subdirectory
- ☑ run the script
- ☑ for the input raster objects choose from the BIGPINE Project File in the SMLDLG directory BAND_3 for Red, BAND_2 for Green, BAND_1 for Blue, and BAND_8 for Pan



Dialog from the ActiveX component program written in Visual Basic and called by VB_PANSHARP.SML

Script Objects and Encryption

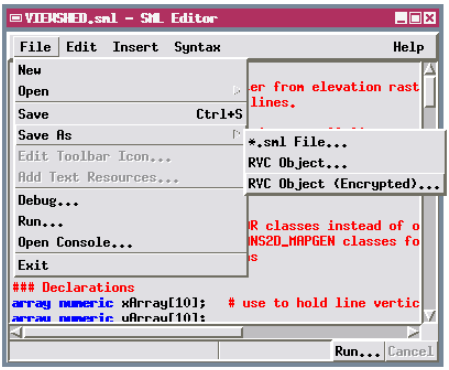
STEPS

- ☑ 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
- ☑ use File / Open / RVC Object to select your encrypted script (the SML window then shows only an encryption message)

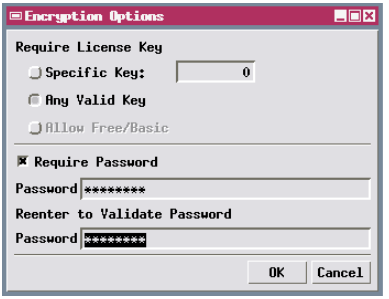
Use the Save As / RVC Object (Encrypted) option to create an encrypted copy of the script in a Project File.

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.



If you open an encrypted script in the SML Editor window, it shows only an encryption message.



IMPORTANT:
Always keep an unencrypted copy of the script for editing.

SML and GeoFormulas

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.

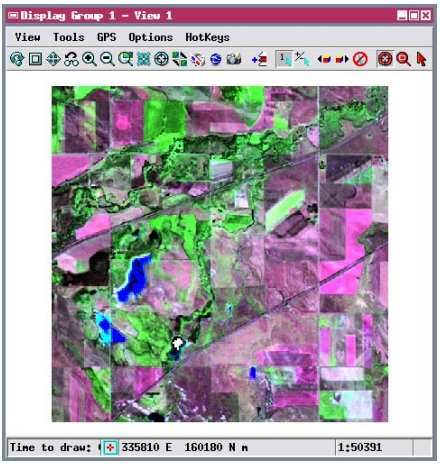
A GeoFormula script can be saved as a reusable file. A GeoFormula layer can be combined with any number of other layers in the TNT display process to create a complex visualization of multiple geospatial objects.

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.

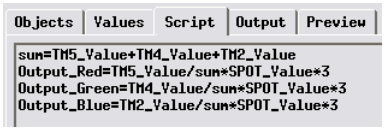
See the tutorial booklet *Using Geospatial Formulas* for a complete introduction to constructing and using GeoFormulas

STEPS

- ☑ choose Main / Display from the TNTmips menu
- ☑ in the Display Manager choose Display / New / Empty 2D
- ☑ choose Add / Layer / Geoformula / Quick-Add Geoformula in the Display Manager
- ☑ select BROV_UMN.GSF from the GEOFRMLA sample data directory
- ☑ 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.



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

TNTmips Basic TNTmips Basic is a low-cost version of TNTmips for small projects.

TNTmips Free TNTmips Free is a free version of TNTmips for students and learning professionals with small projects.

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.

TNTscript TNTscript lets you execute SML scripts on additional computers, at remote sites, or using cloud-computing resources.

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 or DVD at low cost. TNTatlas CDs/DVDs can be used on any popular computing platform.

Index

ActiveX....69
array....31
CAD object....43
coordinate transformation....37-38
database....45-47
debugger....28-29
classes....17-21
encryption....66
functions....15-17,22-24
GeoFormula....71
hash....34
import....54-55
job processing....55
layout....65
LIDAR....52
loops (for, for each, while)....13-14
Macro Script....61
matrix....31
movie script....64
pipeline....56-59
preprocessor commands....28
procedures....15-17,22-24
raster object....39,51
region object....42
Script Builder....59
shape object....44,52
status dialog....53
string....33
stringlist....32
syntax....6,7
TIN objects....43
TNTscript....60
Tool Script....61
user input....25
variable....10,24
vector object....40-41,49-50
view window....61-63

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