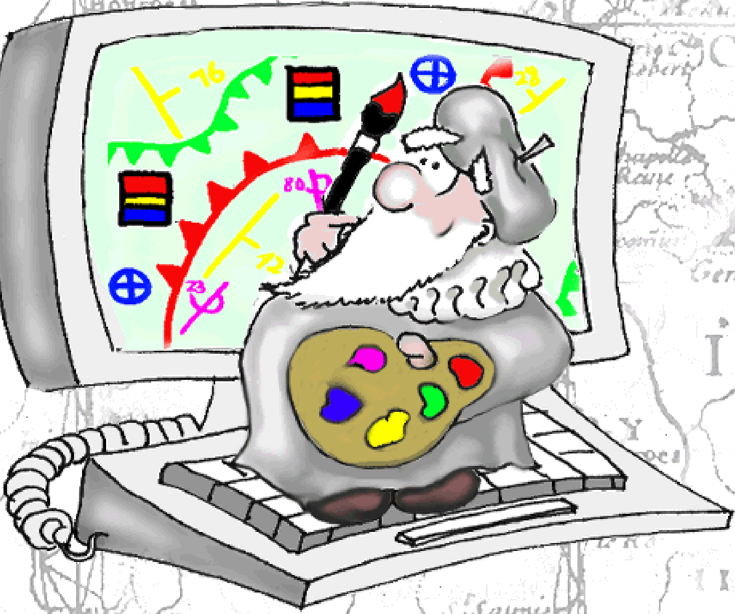
CARTOSCRIPTS

Tutorial

Using CartoScripts™



with

TNTmips®

TNTedit™

TNTview®

Before Getting Started

Some mapping projects may require the use of specialized symbols for lines and points in vector and CAD objects. The cartographic scripting language in TNTmips®, TNTview®, and TNTedit™ provides a complete and flexible set of drawing functions that allow you to design custom map symbols for many applications. CartoScripts extend the symbol-creation capabilities found in the standard point and line style editors in the TNT products. The exercises in this booklet introduce the most commonly-used CartoScript functions, and provide many sample CartoScripts for point and line symbols.

Prerequisite Skills This booklet assumes that you have completed the exercises in the following tutorial booklets: *Displaying Geospatial Data*, *TNT Product Concepts*, *Creating and Using Styles,* and *Building and Using Queries.* Those exercises introduce essential skills and basic techniques that are not covered again here. Please consult those booklets for any review you need.

Sample Data The exercises presented in this booklet use sample data that is distributed with the TNT products. If you do not have access to a TNT products DVD, you can download the data from MicroImages' web site. In particular, this booklet uses the CARTOSMP, TOWNS and GGMAP Project Files in the CARTOSCR data collection. Install the sample files on your hard drive so changes can be saved as you work with them.

More Documentation This booklet is intended only as an introduction to using CartoScripts to style vector or CAD elements. Details of the processes discussed here can be found in a variety of tutorial booklets, Technical Guides, and Quick Guides, which are all available from MicroImages' web site.

TNTmips® Pro and TNTmips Free TNTmips (the Map and Image Processing System) comes in three versions: the professional version of TNTmips (TNTmips Pro), the low-cost TNTmips Basic version, and the TNTmips Free version. All versions run exactly the same code from the TNT products DVD and have nearly the same features. If you did not purchase the professional version (which requires a software license key) or TNTmips Basic, then TNTmips operates in TNTmips Free mode. All of the exercises can be completed in TNTmips Free using the sample geodata provided.

Randall B. Smith, Ph.D., 21 April 2011

You can print or read this booklet in color from MicroImages' web site. The web site is also your source for the newest Getting Started booklets on other topics. You can download an installation guide, sample data, and the latest version of TNTmips Free.

http://www.microimages.com

Welcome to Using CartoScripts

The style editors in TNTmips and TNTview allow you to select, modify, combine, or create a wide variety of standard point and line symbols, as described in the tutorial booklet *Creating and Using Styles*. For those instances in which the standard style editors cannot provide the appropriate symbol, you can use CartoScripts™ to design custom map symbols for points and lines.

CartoScripts are style scripts that utilize special functions in the TNTmips Geospatial Scripting language (in the Cartoscripts function group). The CartoScript functions allow you to draw and navigate along line elements and to draw new lines and shapes to form symbols. Symbols can be repeated along line elements or drawn singly for point elements. You can add labels to the symbols using text from associated database tables and optimize label placement within a single drawing layer. You can also structure a script to use element attributes to vary the symbol styling.

The exercises in this booklet introduce and explain the most commonly-used CartoScript functions and show how you can structure scripts to produce various effects for point and line symbols. CartoScripts are subject to the same syntax rules as standard database queries and SML scripts. For a review of basic query syntax, consult the tutorial booklet *Building and Using Queries*.

Geologic maps are one example of maps requiring specialized point and line symbols that can be drawn using CartoScripts. Point symbols are used to indicate the orientation of outcrop-scale structures, while special line symbols are used to represent map-scale features. Several of the exercises in this booklet use geological examples to illustrate elements of CartoScript structure.

STEPS
- ☑ make sure the sample files mentioned on page 2 have been copied to your hard drive
- ☑ start TNTmips
- ☑ choose Main / Display from the TNTmips menu

The exercises on pages 4-19 illustrate the use of CartoScript functions to create point symbols. Pages 4-8 introduce the basic functions used to draw simple geometric shapes. Pages 9-19 lead you through adding text labels from database fields, drawing more complex symbols, varying symbol orientation by attribute, and optimizing label placement.

The exercises on pages 20-33 show you how to use CartoScripts to create line symbols. Basic line navigation and drawing functions are introduced on pages 20-22. Script structures to create repeated symbols and placement of text labels for lines are explained on pages 23-33.

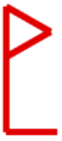
Use of CartoScripts with legends is discussed on pages 34-37, and pages 38-39 provide a complete list of available CartoScript functions.

You can download additional sample CartoScripts for geological point and line symbols from the MicroImages web site:
www.microimages.com / downloads / cartoscripts

Draw Simple Point Symbols

STEPS

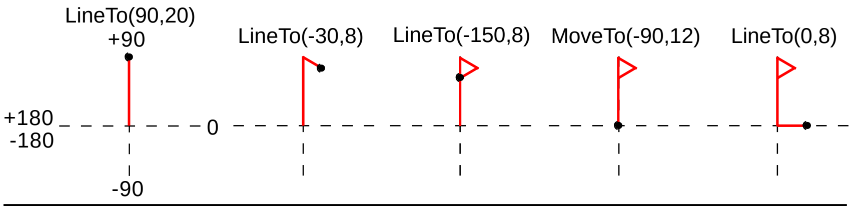
- ☑ click on the Add Objects icon button on the Display Manager window's toolbar
- ☑ navigate to the CARTOSMP Project File in the CARTOSCR data collection and select the SAMPLES vector object
- ☑ click on the Layer Controls icon button in the SAMPLES layer entry in the Display Manager
- ☑ click on the Points tab on the Vector Layer Controls window
- ☑ select By Script from the Style option menu and click [Specify...]
- ☑ press the Open icon button in the Script Editor window and choose Open RVC Object from the menu
- ☑ select object FLAGSCR from the CARTOSMP Project File
- ☑ press [OK] in the Script Editor window and again in the Vector Layer Controls window



The CartoScript you open in this exercise (shown in the box below) draws the flag symbol shown to the right. The first line in the script is a comment line with the script name (remember that comments are preceded by the "#" character). The LineStyleSetColor( ) function in Line 2 sets Red, Green, and Blue values (between 0 and 255) that determine the color of lines drawn by the drawing functions that follow (in this case, red lines). The function in Line 3 sets the line width for the drawing functions (more about this below).

The remaining script lines actually draw the symbol. The LineStyleLineTo( ) function draws a line to the point specified by a direction (first numeric parameter) and distance (second parameter). The LineStyleMoveTo( ) function uses the same sequence of parameters to move the "pen" location without drawing. Both functions reference a local coordinate system centered on the current point element (the script is read and evaluated once for each point element in the object). Directions for these functions are specified by angles (0 to 180 and 0 to -180) relative to the positive ***x*** axis of the object coordinate system. Object coordinates also provide the default units for the distance parameters in these functions, as well as for the width parameter in the LineStyleSetLineWidth( ) function. The sequence of drawing actions is illustrated below, with the dot indicating the pen position at the end of each action.

```
1  ## FlagScr
2  LineStyleSetColor(225,0,0);
3  LineStyleSetLineWidth(1);
4  LineStyleLineTo(90,20);
5  LineStyleLineTo(-30,8);
6  LineStyleLineTo(-150,8);
7  LineStyleMoveTo(-90,12);
8  LineStyleLineTo(0,8);
```

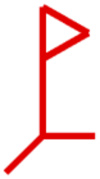


Using Anchors

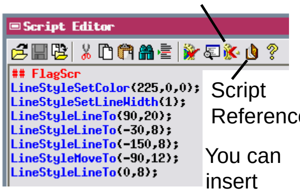
In the sequence of drawing movements in the flag script in the previous exercise, the pen position returns to the origin of the local coordinate system before drawing the final line at the base of the flag. The simple geometry of the flag symbol makes it relatively easy to calculate the angle and distance for the LineStyleMoveTo( ) function in Line 7 that moves the pen to the origin. But you can avoid the need for such calculations by using **anchors**: positions you record for later use in a set of drawing actions. The LineStyleDropAnchor( ) function sets an anchor position and assigns it the number you enter as the numeric parameter for the function. The LineStyleMoveToAnchor( ) function moves the pen to the specified anchor position. There is also a LineStyleLineToAnchor( ) function that draws a line from the current pen position to the specified anchor position. You can establish multiple anchor points to aid in drawing complex symbols, and use them in any order. In the script on this page, we place an anchor at the origin before beginning to draw, and return to the anchor position twice more to draw lines from the base of the flag.

STEPS
- ☑ reopen the Vector Layer Controls window
- ☑ reopen the ScriptEditor window for point styles
- ☑ delete the statement with the LineStyleMoveTo( ) function and insert the statements shown in bold text below
- ☑ press [OK] in the Script Editor window and again in the Vector Layer Controls window

```
LineStyleSetColor(225,0,0);
LineStyleSetLineWidth(1);
LineStyleDropAnchor(1);
LineStyleLineTo(90,20);
LineStyleLineTo(-30,8);
LineStyleLineTo(-150,8);
LineStyleMoveToAnchor(1);
LineStyleLineTo(0,8);
LineStyleMoveToAnchor(1);
LineStyleLineTo(-135,8);
```



Insert Field

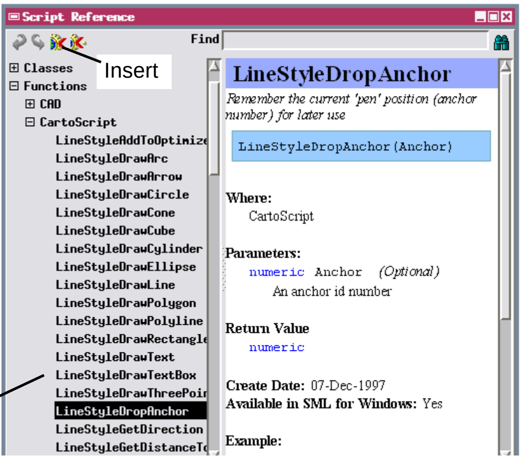


```
## FlagScr
LineStyleSetColor(225,0,0);
LineStyleSetLineWidth(1);
LineStyleLineTo(90,20);
LineStyleLineTo(-30,8);
LineStyleLineTo(-150,8);
LineStyleMoveTo(-90,12);
LineStyleLineTo(0,8);
```

Script Reference

You can insert database field names using the Insert Field icon buton and open the Script Reference window to insert function names, variables, and operators.

Open the Cartoscript function group to see a list of the CartoScript style functions.



LineStyleDropAnchor

Remember the current 'pen' position (anchor number) for later use

`LineStyleDropAnchor(Anchor)`

Where:
CartoScript

Parameters:
numeric Anchor *(Optional)*
An anchor id number

Return Value
numeric

Create Date: 07-Dec-1997
Available in SML for Windows: Yes

Example:

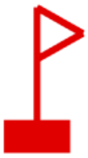
Using Built-in Geometric Shapes

STEPS

- ☑ reopen the Script Editor window for point styles
- ☑ edit the script to duplicate the text below; the new statement is in bold type

```
LineStyleSetColor(225,0,0);
LineStyleSetLineWidth(1);
LineStyleDropAnchor(1);
LineStyleLineTo(90,20);
LineStyleLineTo(-30,8);
LineStyleLineTo(-150,8);
LineStyleMoveToAnchor(1);
LineStyleDrawRectangle(10,5,0,1);
```

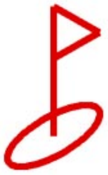
- ☑ press [OK] in the Script Editor window and again in the Vector Layer Controls window



- ☑ reopen the Script Editor window
- ☑ replace the last line in the script with the statement shown below

```
LineStyleDrawEllipse(0,0,8,3,30,0,0);
```

- ☑ press [OK] in the Script Editor window and again in the Vector Layer Controls window



Several functions that draw simple geometric shapes are included in the CartoScript function set. The LineStyleDrawRectangle( ) and LineStyleDrawCircle( ) functions draw their respective shape centered on the current pen position, which is left unchanged. The first script for this exercise draws a filled rectangle at the base of the flag symbol. Four numeric parameters are used in this instance to control the rectangle function: *width*, *height*, *angle*, and *dofill*. The first two parameters are required, and specify the width and height of the rectangle. The third (optional) parameter specifies a rotation angle, which in this example is equal to zero. The last parameter (also optional) determines whether the shape is filled with the current line color (1), or left unfilled (0).

The second example in this exercise draws an unfilled ellipse at the base of the flag symbol. The LineStyleDrawEllipse( ) function has up to 7 parameters: *angle*, *distance*, *radius_x*, *radius_y*, *rotangle*, *isAngleAbs*, and *dofill*; the first four are required. The initial *angle* and *distance* parameters allow you to automatically move the pen to a new position before drawing. In this example, both are set to zero, leaving the ellipse centered on the base of the flag. The pen returns to the ellipse center after drawing. The ellipse dimensions are initially set parallel to the x and y coordinate axes by the two radius parameters (in this example, 8 and 3 units, respectively). The value of 30 for the *angle* parameter rotates the ellipse 30 degrees counterclockwise. The *isAngleAbs* parameter determines whether the ellipse is drawn and rotated relative to the local coordinate system (0), or relative to global object coordinates (1). This distinction does not exist for point data, but becomes important when styling lines, as we will see later.

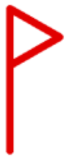
Recording and Drawing Polygons

Styling for the ends of lines is set using the LineStyleSetCapJoinType( ) function, which has *capstype* and *jointype* parameters. The *capstype* parameter draws square line ends when set to 1, or rounded ends when set to 0. The *jointype* parameter uses the same values to determine styling for the ends of segments of a polygon outline or polyline. The default value for both parameters is 1, so squared ends are drawn if you do not include this statement in a script.

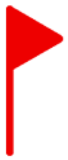
STEPS
- ☑ reopen the Script Editor window for point styles
- ☑ edit the script to duplicate the text below, adding the statement shown in bold type

```
LineStyleSetColor(225,0,0);
LineStyleSetLineWidth(1);
LineStyleSetCapJoinType(0,0);
LineStyleLineTo(90,20);
LineStyleLineTo(-30,8);
LineStyleLineTo(-150,8);
```

- ☑ press [OK] in the Script Editor window and again in the Vector Layer Controls window



- ☑ reopen the Script Editor window
- ☑ change the values for the cap and join parameters as shown, and add the statements shown in bold
- ☑ press [OK] in the Script Editor window and again in the Vector Layer Controls window



- ☑ choose Close from the Layer Manager's Display menu when you have completed this exercise

The script in the second half of this exercise draws a solid-color flag symbol by drawing the triangular flag element as a filled polygon. You can draw simple or complex polygon shapes by using functions to execute the following steps: 1) initiate recording of vertex locations; 2) move to or draw lines to each vertex location in turn; 3) connect the vertices to draw the polygon. The LineStyleRecordPolygon( ) function has a single *start_stop* parameter; a value of 1 starts recording vertex locations specified by pen movements in subsequent statements. The LineStyleDrawPolygon( ) function forms a polygon using the recorded vertices, and has a single *dofill* parameter (set to 1 in this example to fill the triangle). The LineStyleDrawPolygon( ) function also stops the recording of vertex locations, so there is no need to explicitly stop recording with a LineStyleRecordPolygon(0) statement following the vertex movement list. You can also use the same structure to record vertex locations to connect as segments of a single line with the LineStyleDrawPolyline( ) function.

```
LineStyleSetColor(225,0,0);
LineStyleSetLineWidth(1);
LineStyleSetCapJoinType(0,0);
LineStyleLineTo(90,20);
LineStyleDropAnchor(2);
LineStyleRecordPolygon(1);
LineStyleLineTo(-30,8);
LineStyleLineTo(-150,8);
LineStyleMoveToAnchor(2);
LineStyleDrawPolygon(1);
```

Using 3D Shapes

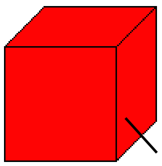
STEPS

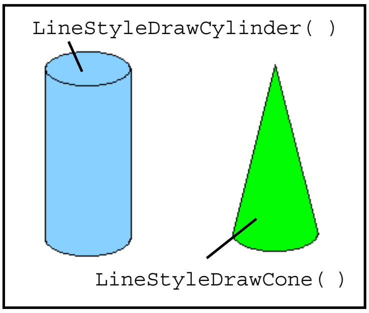
- ☑ choose Display / Open from the Layer Manager
- ☑ select the CubeGroup object from the CARTOSMP Project File
- ☑ open the Vector Layer Controls window for the SAMPLES layer
- ☑ open the Script Editor window for point styles and examine the script CubeScr1
- ☑ press [OK] in the Script Editor window and again in the Vector Layer Controls window

CartoScript functions are also available to draw perspective renderings of simple three-dimensional shapes: a rectangular solid, vertical cylinder, and vertical cone. Edge lines in each symbol are drawn using the color specified by the LineStyleSetColor( ) function. Three color parameters for each function set the red, green, and blue values for the fill color. The cube symbol is drawn with the base of the front face centered on the current pen position. The cylinder and cone symbols are drawn so that the center of the basal ellipse coincides with the current pen position.

The *width*, *depth*, and *height* parameters for the LineStyleDrawCube( ) function specify the lengths of the corresponding edges of the rectangular solid. If you want the symbol to appear in perspective as a true cube, as in this exercise, the *width* and *height* should be equal and the *depth* value should be about half the length of the other dimensions.

You can define numeric or string variables in a script for later use as function parameters. Defining variables (with comments) at the beginning of a script makes it easier to find and edit necessary parameter values when you are reusing and modifying a script.





```
# CubeScr1
# Set dimensions of cube symbol
numeric width = 15;
numeric depth = 0.5 * width;
numeric height = width;

# Set color for cube faces
numeric red, green, blue;
red = 255; green= 0; blue = 0;

# Draw cube symbol
LineStyleSetColor(0,0,0);      # line color for edges
LineStyleSetLineWidth(0);
LineStyleDrawCube(width,depth,height,red,green,blue);
```

Text Labels from Database Fields

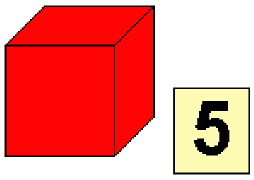
The script in this exercise adds a boxed text label to the cube symbol. The additional statements needed to format and draw the label are shown below. The label is a sample number stored as a numeric value in a database field. In order to use it as a label, the number must first be converted to a text string and assigned to a string variable using the sprintf( ) function. The first parameter in this function is a string (in quotes) with formatting information; “%d” indicates an integer value. The second parameter in this case is the location of the numeric value, specified in the form TableName.FieldName.

Text labels are drawn with the lower left corner corresponding to the pen position. The orientation of a label is set by the *angle* parameter. The *border* parameter specifies the width of the border between the text label and its surrounding box. The last parameter shown here, *isAbs*(0), indicates the reference frame of the *angle* parameter.

STEPS
- ☑ reopen the Vector Layer Controls windowfor the SAMPLES layer
- ☑ open the Script Editor Editor window for point styles
- ☑ press the Open icon button and choose Open RVC Object
- ☑ select CUBESCR2 in the CARTOSMP Project File
- ☑ press [OK] in the Script Editor Editor window and again in the Vector Layer Controls window

You should get in the habit of terminating each statement in a script with a semicolon (;). This helps the syntax checker pinpoint the location of syntax errors.

```
# Read sample number from database field
# and convert to text string for use as a label
string label$ = sprintf("%d",Samples.Number);

# String variable for label text font
string font$ = "ARIALBD.TTF";

# Define color variables for text
numeric tred, tgreen, tblue;
tred = 0;     tgreen= 0;    tblue = 0;
# Define fill color variables for text box
numeric fillred, fillgreen, fillblue;
fillred = 255;       fillgreen = 255;      fillblue = 170;
# Define height, angle, and border width of text box
numeric t_height, angle, border;
t_height = 10;       angle = 0; border = 2;
# Set color and font for text label
LineStyleSetTextColor(tred,tgreen,tblue,fillred,
       fillgreen,fillblue);
LineStyleSetFont(font$);

# Move pen to right of symbol and draw label
LineStyleMoveTo(0, width * 1.2);
LineStyleDrawTextBox(label$,t_height,angle,border,0);
```



Setting Coordinate Type Options

STEPS

☑ zoom in and out and note the effect on the relative sizes of point symbols and contours

☑ reopen the Vector Layer Controls window for the SAMPLES layer and Script Editor window for point styles

☑ press the Open icon button and choose Open RVC Object

☑ select CUBESCR3 from the CARTOSMP Project File

```
#  CubeScr3
#  Set  dimensions  of  cube  symbol
LineStyleSetCoordType(1);
numeric  width  =  4;
numeric  depth  =  0.5  *  width;
numeric  height  =  width;
```

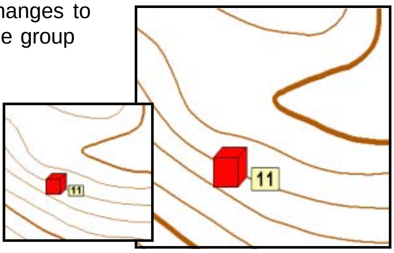
☑ press [OK] in the Script Editor window and again in the Vector Layer Controls window

☑ zoom in and out and note the effect on the relative sizes of the symbols and contour lines

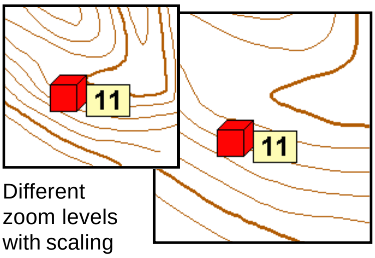
☑ choose Display/Close when you have completed this exercise, and click [No] on the dialog when you are asked whether to save changes to the group

The default units for the size and distance values you use in CartoScripts are in internal object coordinates (meters for the objects you have used in these exercises). Thus as you change zoom levels, the size of the symbols on the screen change as the display scale changes, maintaining a constant size in object coordinates. The script in this exercise adds the statement shown in bold type to the previous CubeScr2 (along with changes in the symbol and label size values). A parameter value of 1 for the LineStyleSetCoordType( ) function changes the size and distance units to millimeters at the current display scale or print scale. When you change zoom levels or print scales, symbols maintain a constant display size in millimeters. This setting works best with layouts designed only for display (A value of 0 for this parameter is equivalent to the default condition).

If you are using CartoScripts to draw map elements designed for printing at a particular scale (such as 1:24,000), you can use the default coordinate setting to maintain the relative sizes of different elements at different zoom levels for screen display, but use the map scale to compute the element sizes needed to produce the desired sizes on the final printed map. See the script on pages 14-15 for an example of this scaling approach.



Different zoom levels with default scaling to object coordinates. The point symbol has a constant size in object coordinates.



Different zoom levels with scaling to millimeters. The point symbol maintains a constant size in screen (or print) coordinates.

3D Cylinder Bar Graph

The 3D symbols can be combined together to form 3D bar graphs, with the height of each bar determined by the value in a database field. You can change the perspective of the cylinder and cone symbols by varying the relative lengths of the long and short axes of the cylinder and cone base.

The LineStyleTextNextPosition( ) function is used in this script to approximately center the "Cu" element label below its cylinder. The first four parameters of this function specify the string, its height, angle, and local or absolute coordinate reference. The last three parameters, *next_x*, *next_y*, and *length*, are variables which the function creates to hold the x-coordinate, y-coordinate, and length of the label string. These values can be used in subsequent statements to guide positioning.

STEPS

☑ click on the Add Objects icon button on the Display Manager window's toolbar

☑ select the GEOCHEMVEC object from the CARTOSMP Project File

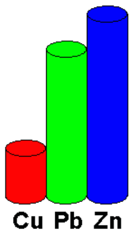
☑ set point styling to By Script and open the Script Editor window

☑ press the Open icon button and choose Open RVC Object

☑ select object CYLINGRAPH in the CARTOSMP Project File

☑ press [OK] in the Script Editor window and again in the Vector Layer Controls window

```
# Read values from database and assign to
# variables to use for cylinder heights
numeric cuVal = Geochemistry.Cu;
numeric pbVal = Geochemistry.Pb;
numeric znVal = Geochemistry.Zn;
# Set edge line color and cylinder dimensions
LineStyleSetColor(0,0,0);
numeric longAxis = 120;  numeric  shortAxis = 50;
# Set text color and font
LineStyleSetTextColor(0,0,0);
LineStyleSetFont("ARIALBD.TTF");
# Draw three cylinders side by side
LineStyleDropAnchor(1);
LineStyleDrawCylinder(longAxis,shortAxis,cuVal,255,0,0);
LineStyleMoveTo(0,longAxis);
LineStyleDrawCylinder(longAxis,shortAxis,pbVal,0,255,0);
LineStyleMoveTo(0,longAxis);
LineStyleDrawCylinder(longAxis,shortAxis,znVal,0,0,255);
# Draw label centered below each cylinder
numeric next_x, next_y, length;
LineStyleMoveToAnchor(1);      # move to base of first cylinder
LineStyleMoveTo(-90,100);      # move down to make room for label
LineStyleTextNextPosition("Cu",70,0,0,next_x,next_y,length);
LineStyleMoveTo(180,length * 0.4);  # move label point to left
LineStyleDrawText("Cu",70,0,0);     # to center first label
LineStyleMoveTo(0,longAxis);     # move right by width of cylinder
LineStyleDrawText("Pb",70,0,0);
LineStyleMoveTo(0,longAxis);
LineStyleDrawText("Zn",70,0,0);
```

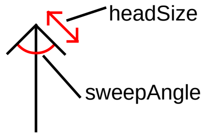


☑ choose Display / Close when you have completed this exercise

Orienting Symbols by Attribute

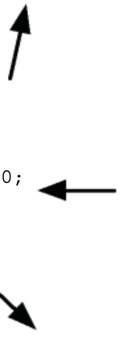
STEPS

- ☑ click on the Add Objects icon button in the Display Manager window
- ☑ select the ArrowPts object from the cartosmp Project File
- ☑ set point styling to By Script and open the Script Editor window
- ☑ press the Open icon button and choose Open RVC Object
- ☑ select ArrowScr1 in the cartosmp Project File
- ☑ press [OK] in the Script Editor window and again in the Vector Layer Controls window



You can use a script to draw symbols that vary in orientation depending on a direction value read from a database field. In this instance the arrow directions are in azimuth form (0 to 360° angle measured clockwise from north), and must be converted to the internal coordinate system used by the drawing functions. This script also converts all negative direction values to the corresponding positive values, but this conversion is not required.

The LineStyleDrawArrow( ) function draws arrowheads bounded by straight lines whose length is defined by the *headSize* variable in this script. The angle between these bounding lines is set by the *sweepAngle* parameter. If the *dofill* parameter is set to 1, the head is filled to form a solid triangle. The function updates the pen position to the tip of the arrow head. Arrow heads do not render well with a line width larger than 0, so this script draws the arrow with 0 line width, then redraws the arrow stem with a wider line.



```
# Read azimuth from database table
azim = Direct.Azimuth;
# Convert azimuth to internal coordinate system
direction = -(azim - 90);
    if (direction < 0) then  direction = direction + 360;
# Set color values for symbol
red, green, blue; red = 0;    green = 0; blue = 0;
LineStyleSetColor(red,green,blue);
# Set dimensions for arrow
arrowLength = 30;
headSize = 0.4 * arrowLength;
sweepAngle = 40; dofill = 1;
# Draw arrow with zero line width, tip of stem at point
LineStyleDropAnchor(0);       # anchor at point
LineStyleSetLineWidth(0);
LineStyleDrawArrow(direction,arrowLength,headSize,sweepAngle,dofill);
LineStyleDropAnchor(1);       # anchor at tip of arrow
# Redraw arrow stem with wider line
stemLength = arrowLength - headSize * cosd(sweepAngle);
LineStyleMoveToAnchor(0);
LineStyleSetLineWidth(1.5);
LineStyleLineTo(direction,stemLength);
LineStyleMoveToAnchor(1);# pen to arrow tip in prep for label
```

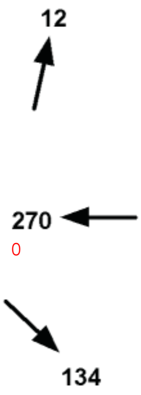
Calculating Label Positions

The script in this exercise adds a label with the azimuth to each arrow symbol from the previous exercise. The additional statements needed to format and draw the labels are shown below. The tricky part is varying the label position based on the arrow orientation to avoid overlap between the arrow and its label. The script computes two pen shifts that are applied before the label is drawn, as explained in the comments. Each is computed as a function of the direction angle and the height and length of the label string. The second shift must be computed separately for each quadrant, being careful that the resulting distance is a positive value (values for distance parameters of drawing functions must be positive; a negative value is interpreted as 0).

STEPS

- ☑ reopen the Script Editor window for point styles
- ☑ press the Open icon button and choose Open RVC Object
- ☑ select ArrowScr2 in the cartosmp Project File
- ☑ press [OK] in the Script Editor window and again in the Vector Layer Controls window
- ☑ choose Display / Close from the Layer Manager when you have completed this exercise

```
# Convert azimuth value to text string for use as a label
label$ = sprintf("%d",azim);
# Find length of label text for label positioning
height = 10;
LineStyleTextNextPosition(label$,height,0,0,next_x,next_y,length);
# Set font name and color
LineStyleSetFont("ARIALBD.TTF");
LineStyleSetTextColor(red,green,blue);
# Compute label shift perpendicular to arrow to center label
shift1 = 0.5 * (height*cosd(direction) - length*sind(direction));
# Compute label shift parallel to arrow to avoid overwriting arrow
offset = arrowLength * 0.1;
if (direction >= 0 and direction < 90) then
    shift2 = offset;
else if (direction >= 90 and direction < 180) then
    shift2 = offset - length * cosd(direction);
else if (direction >= 180 and direction < 270) then
    shift2 = offset - length * cosd(direction)
        - height * sind(direction);
else if (direction >= 270 and direction <= 360) then
    shift2 = offset - height * sind(direction);
# shift pen position and draw label
if (shift1 < 0) { # MoveTo distances can't be less than 0
    shift1 = abs(shift1); # absolute value
    LineStyleMoveTo(direction + 90,shift1); }
    else LineStyleMoveTo(direction - 90,shift1);
LineStyleMoveTo(direction,shift2);
LineStyleDrawText(label$,height,0,0);
```

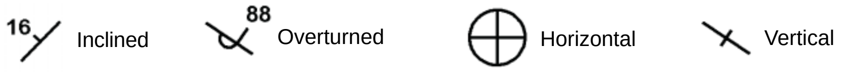


Drawing Strike and Dip Symbols

STEPS

- ☑ click on the Add Objects icon button in the Display Manager window
- ☑ select the BEDDING object from the CARTOSMP Project File
- ☑ set point styling to By Script and open the Script Editor window
- ☑ press the Open icon button and choose Open RVC Object
- ☑ select BEDDINGSCR in the CARTOSMP Project File
- ☑ press [OK] in the Script Editor window and again in the Vector Layer Controls window
- ☑ select Close from the Display Manager's Display menu when you have completed this exercise

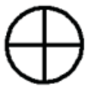
Geologic maps use special point symbols to indicate the orientation of planar or linear structures in rock outcrops. The script in this exercise draws standard symbols showing the strike and dip of layering (bedding) in sedimentary rocks, as shown below. It uses many of the functions and concepts introduced previously. The dip and strike are read from an associated database table. (The strike angle must be specified as an azimuth using the so-called right-hand rule: the strike line points toward the azimuth for which the dip direction is to the right.) The symbol is oriented so that the long (strike) line is parallel to the strike direction, and the symbol is labeled with the value of the dip angle. Special symbols are drawn for horizontal and vertical beds (special values of the dip angle), and for overturned beds (indicated by a logical field in the database). The second half of the script, which labels the symbols with the dip value, resembles the script on the previous page, and is not shown here.

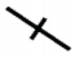


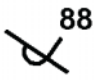
```
# Read strike azimuth and dip value from table.field
azStrike = Bedding.Strike - northAng;   dip1 = Bedding.Dip;
# Check logical field for overturned bedding.  Variable is set
# to 1 if Yes, 0 if No
overturned = Bedding.Overturned;
# Variables define the color of the symbols and label
red = 0;      green = 0;     blue = 0;
# This variable defines the denominator of the intended map scale.
scale = 5000;
# These variables define the dimensions and line widths of the
# symbol.  strikeLengthMap is the desired length of the symbol
# strike line in mm, assuming vector coordinates are in meters.
# lineWidthMap is the desired line width in mm.
strikeLengthMap = 6;      lineWidthMap = 0.3;
strikeLength = strikeLengthMap * scale / 1000;
halfLength = 0.5 * strikeLength;
tickLength = halfLength / 3;
doubTick = tickLength * 2;
lineWidth = lineWidthMap * scale / 1000;
```

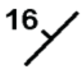
Strike and Dip Script (continued)

```
###### Process
# Convert strike azimuth to internal coordinate system.
direction = -(azStrike - 90);
if (direction < 0) then
   direction = direction + 360;
oppStrike = direction -180;
dipDir = direction - 90;
oppDip = dipDir - 180;

# Set line color, width, and end type
LineStyleSetColor(red, green, blue);# set symbol color
LineStyleSetLineWidth(lineWidth);
LineStyleSetCapJoinType(1,1);    # square ends of lines

########### Draw symbol
# Special symbol for horizontal bedding (cross in circle)
if (dip1 == 0){
   LineStyleDropAnchor(0);
   LineStyleDrawCircle(halfLength);
   LineStyleMoveTo(90, halfLength);
   LineStyleLineTo(-90, strikeLength);
   LineStyleMoveToAnchor(0);
   LineStyleMoveTo(0, halfLength);
   LineStyleLineTo(180, strikeLength);
   }
else {
   # For nonzero dip, draw strike line with center at point
   LineStyleDropAnchor(0);
   LineStyleMoveTo(direction, halfLength);
   LineStyleLineTo(oppStrike, strikeLength);
   LineStyleMoveToAnchor(0);

   # Draw appropriate symbol for dip direction
   if (dip1 == 90) {             # crossbar for vertical bed
      LineStyleMoveTo(dipDir, tickLength);
      LineStyleLineTo(oppDip, doubTick);
      }
   else {
      if (overturned == 1) { # dip symbol for overturned beds
         LineStyleDrawArc(0, 0, tickLength, tickLength,
                              direction, -180, 0);
         LineStyleMoveTo(direction, tickLength);
         LineStyleLineTo(oppDip, doubTick);
         }
      else {                     # dip direction tick mark
         LineStyleLineTo(dipDir, tickLength);
         }
      }
   }
```









Label Optimization

STEPS

- ☑ choose Display / Open from the Layer Manager
- ☑ select the OPTGROUP object from the TOWNS Project File
- ☑ open the Vector Layer Controls window for the TOWNS layer
- ☑ open the Script Editor window for point styles and examine the script OPTSCR1
- ☑ press [OK] in the Script Editor window and again in the Vector Layer Controls window

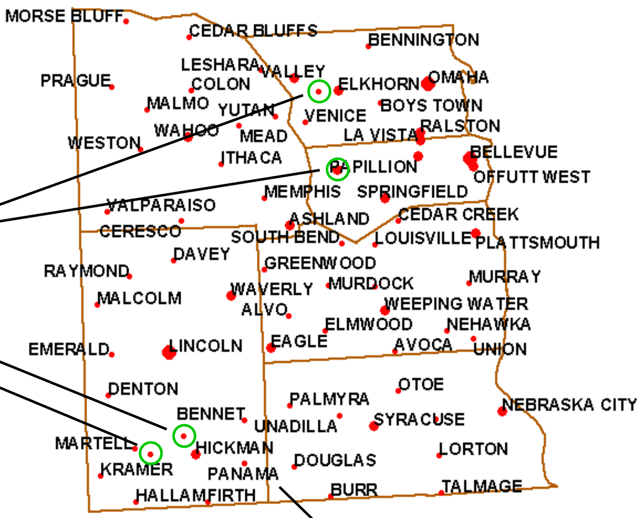
Label Optimization is a procedure for automatically finding the optimal set of positions for point symbol labels generated from a database field. The goal is to avoid overprinting one label with a nearby symbol's label. Individual labels can be moved or deleted to avoid these collisions. The optimizer can automatically place a label in one of a number of different positions around the symbol. The preferred position places the lower left corner of the label on the point. Points can be ranked using attribute information, and these rankings can be used by the optimizer to give preference to higher-ranking points when moving or deleting labels.

The display group used in this exercise shows a block of counties in eastern Nebraska and the included towns and cities. In the script on the facing page the 1990 populations of the towns are used to assign each to one of three *rank* values. The rankings are used to draw point symbols of different sizes, and they are also used by the label optimizer to select labels for deletion.

A CartoScript is executed once for each element in an object, yet label optimization requires information about all labels to resolve positioning conflicts. To solve this dilemma, optimization scripts use the LineStyleAddToOptimizer( ) function to collect information about the dimensions of each label. After all points have been processed, the optimizer moves or deletes labels as needed, then calls a function called FuncDrawLabel( ) to draw the labels. The instructions for this function must be included in a function definition in the script, as shown at the bottom of the facing page. This definition should specify the font, color, and height of the label, and include the LineStyleDrawText( ) or LineStyle-DrawTextBox( ) function.

The first four parameters of the LineStyleAdd-ToOptimizer( ) function are used to determine the dimensions of a label's bounding rectangle. The *xstart* and *ystart* parameters set the lower left corner of the label, and can be read from the internal object coordinates as shown. The *xlast* and *ylast* parameters, which set the upper right corner, can be calculated from the label height and the *length* parameter returned by the LineStyleTextNext-Position( ) function. A value of 0 for the *dooptimize* parameter limits changes in label position to a single pass through the point labels. The final *dodelete* parameter is used to turn label deletion on (1) or off (0). With deletion on, a conflicting label of equal or lower rank may be deleted during optimization.

Optimization with Ranked Deletion



Display of group with ranked deletion. Circles indicate towns for which the label was deleted during optimization.

There are some random aspects to label position changes during optimization, so redrawing the group may cause some labels to change position and previously deleted labels to reappear. The illustration above shows one possible set of label positions.

```
# Rank towns by population
pop = TownData.POP90;
if (pop < 1000) then rank = 1;
else {
    if (pop >= 100000 ) then rank = 3;
    else rank = 2;
    }
# Draw circle with size based on rank
radius = rank * 500;
LineStyleSetColor(255,0,0);
LineStyleDrawCircle(radius,1);
# Read label text and determine dimensions
height = 3500;
LineStyleSetFont("ARIALBD.TTF");
LineStyleTextNextPosition(TownData.NAME,height,0,0,next_x,
                next_y,length);
# Define parameters for optimization
xstart = Internal.x;        ystart  = Internal.y;
xlast = xstart + length;    ylast = ystart + height;
dooptimize = 0;             dodelete = 1;
LineStyleAddToOptimizer(xstart,ystart,xlast,ylast,rank,
                dooptimize,dodelete);
# Define required function to draw labels after optimization
func FuncDrawLabel() {
    LineStyleSetFont("ARIALBD.TTF");
    LineStyleSetTextColor(0,0,0);
    height = 3500;
    LineStyleDrawText(TownData.NAME,height,0,0);
    }
```

Full Label Optimization

STEPS

☑ open the Script Editor window for point styles

☑ press the Open icon button and choose Open RVC Object

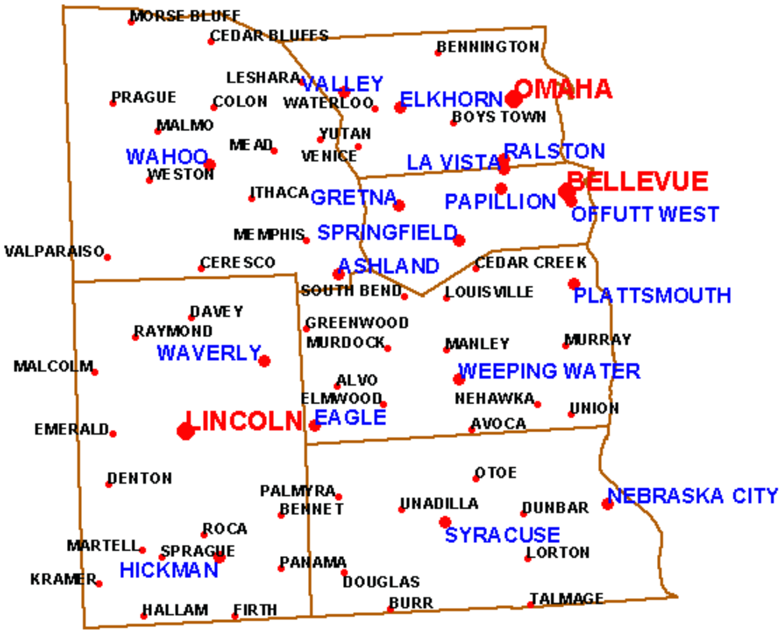
☑ choose OptScr2 from the towns Project File

☑ press [OK] in the Script Editor window and again in the Vector Layer Controls window

☑ close the current display group when you have completed this exercise, and click [No] on the dialog when you are asked whether to save changes to the group

Full label optimization is enabled by setting the value of the *dooptimize* parameter in the LineStyleAddToOptimizer( ) function to 1. The optimizer then makes multiple passes through the point labels to determine optimal label positions. The script on the facing page uses full optimization without deletion to place labels. It also assigns different label sizes and colors for towns of different rank, as illustrated below.

Note that the FuncDrawLabel( ) function cannot directly access any of the variable values assigned in the main body of the script (including the *rank* variable used by the optimizer). In order to vary the label drawing style by rank, as in this script, the FuncDrawLabel( ) function declaration must repeat the ranking procedure found in the main body of the script (as well as the font assignment, source of the label text, and other label attributes).



Full Optimization Script

```
#  Label  sizes  for  three  sizes  of  towns
small = 2500;  med = 3500;   big = 4500;

#  Rank  towns  by  population
pop = TownData.POP90;
if  (pop  <  1000)  then  {
    rank = 1;  height = small;
    }
else  {
    if  (pop  >=  30000  )  then  {
        rank = 3;  height = big;
        }
    else  {
        rank = 2;  height = med;
        }
    }
#  Draw  circle  with  size  based  on  rank
radius = rank * 500;
LineStyleSetColor(255,0,0);
LineStyleDrawCircle(radius,1);

#  Read  label  text  and  determine  dimensions
LineStyleSetFont("ARIALBD.TTF");
LineStyleTextNextPosition(TownData.NAME,height,0,0,next_x,
                    next_y,length);
#  Define  parameters  for  optimization
xstart = Internal.x;      ystart  = Internal.y;
xlast = xstart + length;  ylast = ystart + height;
dooptimize = 1;       dodelete = 0;
LineStyleAddToOptimizer(xstart,ystart,xlast,ylast,rank,
                    dooptimize,dodelete);

#  Define  required  function  to  draw  labels  after  optimization
func  FuncDrawLabel()  {
    small = 2500;   med = 3500;   big = 4500;
    pop = TownData.POP90;
    if  (pop  <  1000)  {
        height = small;  LineStyleSetTextColor(0,0,0);
        }
    else  {
        if  (pop  >=  30000  )  {
            height = big;   LineStyleSetTextColor(255,0,0);
            }
        else  {
            height = med;   LineStyleSetTextColor(0,0,255);
            }
        }
    LineStyleSetFont("ARIALBD.TTF");
    LineStyleDrawText(TownData.NAME,height,0,0);
    }
```

CartoScript Shortcut

Any point symbol that you design using the Point Style Editor can also be saved as a CartoScript, complete with declared variables and comments. You can use this shortcut to create the basic script structure, then add any custom features or references to database fields as needed.

Contour Line Width by Script

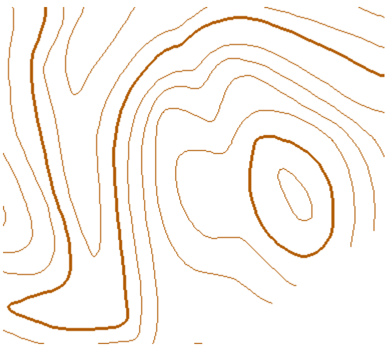
STEPS

- ☑ click on the Add Objects icon button in the Display Manager window
- ☑ select object CONTOURS from the CARTOSMP Project File
- ☑ open the Vector Layer Controls window and click on the Lines tab
- ☑ select By Script from the Style option menu and click [Specify...]
- ☑ press the Open icon button in the Script Editor and choose Open RVC Object
- ☑ select CONSCR from the CARTOSMP Project File
- ☑ press [OK] in the Script Editor window and again in the Vector Layer Controls window
- ☑ use the Zoom Box tool to zoom in on the area of closed contours near the bottom of the object

CartoScripts can also be used to draw simple or complex symbols for line elements in vector or CAD objects. The most basic function for drawing line symbols is the LineStyleDrawLine( ) function, which has no parameters. It simply draws a solid line for each line element using the color set by the LineStyleSetColor( ) function and the width set by the LineStyleSetLineWidth( ) function. It returns the pen position to the beginning of the line after drawing.

The script in this exercise draws lines with different widths for major and minor elevation contours. The original map has a contour interval of 40 feet, and every fifth contour (evenly divisible by 200 feet) is a major contour shown by a wider line. In TNTmips, however, internal Z-values are stored in meters. The script reads the minimum z-value for the line from the Internal table and converts it to feet. The elevation in feet is then divided by 200 using the modulo operator, which returns the remainder of the division, stored here in the variable *rem*. The value of *rem* is 0 only for major contours, so this value is used to assign the appropriate width before drawing the line.

```
# Read contour elevation & convert to feet
elevm = Internal.MinZ;
elevft = round(elevm * 3.28084);
           # Use modulo operator to
           # identify contour elevations
           # that are not evenly divisible
           # by 200 (nonzero remainder)
           rem = elevft % 200;
           # Define widths for minor and
           # major contours
           if (rem <> 0) then width = 2;
              else width = 6;
           # Set line color and width and
           # draw line
           LineStyleSetColor(170,85,0);
           LineStyleSetLineWidth(width);
           LineStyleDrawLine();
```



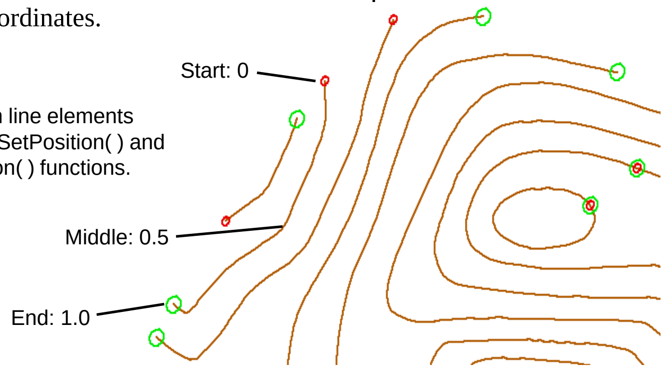
Navigating Lines

Other drawing functions can be used in conjunction with the LineStyleDrawLine( ) function to create more complex line symbols. The script in this exercise first draws the line element as a solid line, then draws circles of different size and color at each end of the line. This script could be used for vector line elements you are editing in the Spatial Data Editor.

A number of functions are provided to allow you to navigate along a line element in order to draw symbol components. The LineStyleSetPosition( ) function is used in this script to move the pen position to the end of each line after marking the start. The single numeric parameter of this function specifies a line position as a relative distance between 0 (beginning of the line) and 1.0 (end of the line). You can use the LineStyleGetPosition( ) function to find the current pen position and assign the value returned to a variable. Set this function's single parameter to 0 if you want the returned value to be the relative position. If you set it to 1, the value returned is the absolute distance from the start in object coordinates.

STEPS
- ☑ reopen the Script Editor window for line styles
- ☑ press the Open icon button and choose Open RVC Object
- ☑ select StartEndScr from the cartosmp Project File
- ☑ press [OK] in the Script Editor window and again in the Vector Layer Controls window

```
# Set radius parameters for small and
# large circles
radius1 = 8; radius2 = 16; dofill = 0;
# Set line color and width and draw line
LineStyleSetColor(170,85,0);
LineStyleSetLineWidth(4);
LineStyleDrawLine();
# Draw small red circle at start of line
LineStyleSetColor(225,0,0);
LineStyleDrawCircle(radius1,dofill);
# Move to end of line and draw large
# green circle
LineStyleSetPosition(1);
LineStyleSetColor(0,225,0);
LineStyleDrawCircle(radius2,dofill);
```



Relative position on line elements using the LineStyleSetPosition( ) and LineStyleGetPosition( ) functions.

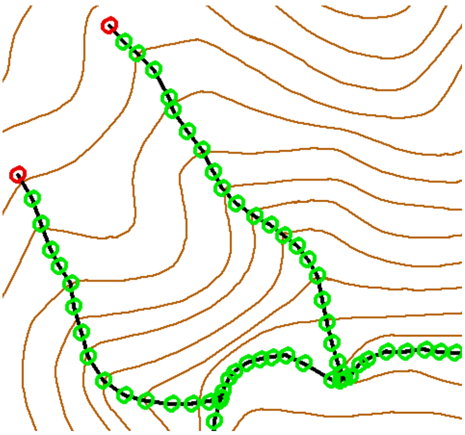
Marking Line Vertices

STEPS

- ☑ click on the Add Objects icon button in the Display Manager window
- ☑ select object STREAMS from the CARTOSMP Project File
- ☑ open the Vector Layer Controls window and click on the Lines tab
- ☑ select By Script from the Style option menu and click [Specify...]
- ☑ press the Open icon button in the Script Editor and choose Open RVC Object
- ☑ select VERTEXSCR from the CARTOSMP Project File
- ☑ press [OK] in the Script Editor window and again in the Vector Layer Controls window

CartoScripts can also mark line vertices as an aid to editing. The script in this exercise draws a red circle at the beginning of each line element, and a green circle at each subsequent vertex.

The LineStyleNextVertex( ) function used in this script moves the pen position to the next vertex along the line. The function also returns a value of 1 if the end of the line has been reached, or 0 otherwise. (The LineStylePrevVertex( ) function moves to the previous vertex, and returns a value of 1 at the beginning of the line.) These functions can thus be used in a “while” loop structure to repeat a set of drawing actions at each vertex. In this script, the “while” loop repeats as long as the LineStyleNextVertex( ) function returns a value of 0. The loop terminates when the function returns a value of 1 at the end of the line.



```
# Set parameters for circles
# marking vertices
radius = 5;       dofill = 0;

# Draw solid black line
LineStyleSetLineWidth(3);
LineStyleSetColor(0,0,0);
LineStyleDrawLine();

# Draw red circle at beginning of line
LineStyleSetColor(225,0,0);
LineStyleDrawCircle(radius,dofill);

# While not at end of line, move to
# next vertex and draw green circle
LineStyleSetColor(0,225,0);
while (LineStyleNextVertex() <> 1) {
    LineStyleDrawCircle(radius,dofill);
    }
```

- ☑ when you have completed this exercise, turn off the Show / Hide checkbox for the STREAMS layer entry in the Display Manager to hide to hide the layer

Drawing Regularly Spaced Symbols

Line symbols you create for use in map layouts may include components spaced regularly along each line element. This exercise illustrates the basic structure of such a script by drawing filled circles equally spaced along the lines.

The LineStyleRoll( ) function moves a specified distance along a line element without drawing. The distance to move is set by the value of the single function parameter. The function also returns a value of 0 for any line position except the end, where it returns a value of 1. By checking this returned value you can use the function in a "while" loop structure to repeat a set of drawing actions at regular intervals along each line.

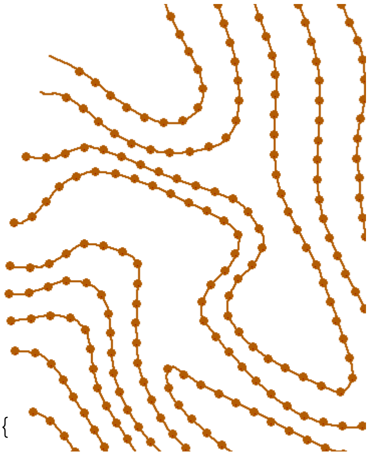
This script is structured to continue drawing circles as long as the distance from the current position to the end of the line is greater than the desired spacing. The LineStyleGetDistanceTo( ) function is used to check this distance. The distance to various line features can be determined by setting the appropriate parameter value for this function, as shown in the box to the right.

```
# Set parameters for circles
radius = 6;        dofill = 1;
# Set spacing between circles
spacing = 30;

# Set line color and width and draw line
LineStyleSetColor(170,85,0);
LineStyleSetLineWidth(3);
LineStyleDrawLine();

# Draw circle at start of line
LineStyleDrawCircle(radius,dofill);

# Draw rest of circles
while (LineStyleRoll(spacing) <> 1) {
    dist = LineStyleGetDistanceTo(3);
    if ( dist > spacing) {
        LineStyleDrawCircle(radius,dofill);
        }
    }
```

STEPS
- ☑ reopen the ScriptEditor window for line styles for the CONTOURS object
- ☑ press the Open icon button and choose Open RVC Object
- ☑ select CIRCLINESCR from the CARTOSMP Project File
- ☑ press [OK] in the Script Editor window and again in the Vector Layer Controls window

Parameter values for the LineStyleGetDistanceTo( ) function:

1 =	next vertex
2 =	previous vertex
3 =	end of line
4 =	beginning of line



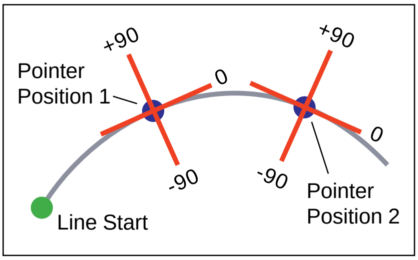
Positions and Coordinate Systems

STEPS

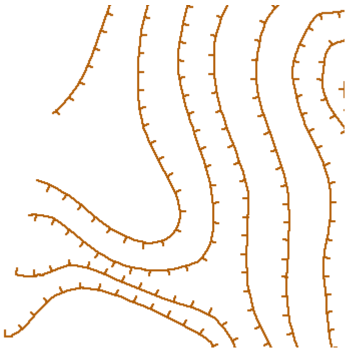
- ☑ reopen the Script Editor window for line styles for the CONTOURS object
- ☑ press the Open icon button and choose Open RVC Object
- ☑ select TickLineScr from the CARTOSMP Project File
- ☑ press [OK] in the Script Editor window and again in the Vector Layer Controls window

The CartoScript drawing engine keeps track of two positions during execution of a script. The first is the current position along a line element, which you can think of as a "pointer" that is moved along the line by the LineStyleRoll( ) function or the other line navigation functions. The second position that is tracked is the pen position, which may or may not be on the line element.

The current pointer position serves as the origin of a local coordinate system that is oriented relative to the line element as shown in the boxed illustration. The drawing functions that use angle and distance parameters to move the pen position or draw elements, such as LineStyleLineTo( ) and LineStyleDrawArrow( ), reference this local coordinate system. This system enables you to draw repeated symbol components that are oriented consistently relative to the local line direction, such as the perpendicular tick lines in the script in this exercise. The tick lines are drawn on the left side of each line (relative to the start and end points). If asymmetric symbols such as this need to be drawn on a particular side of each line, you may need to use the TNTmips spatial data Editor to reverse the direction of individual lines in order to create the desired symbol.



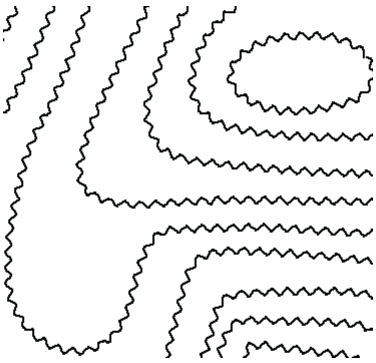
```
# Denominator of
# desired map scale
scale = 5000;
# Desired spacing and
# length of tick lines
# in mm at map scale
spaceMap = 6; lengthMap = 1.5;
# Scaled spacing and length
# of tick lines
spacing = spaceMap * scale / 1000;
length = lengthMap * scale / 1000;
# Set line color and width and draw line
LineStyleSetColor(170,85,0);
LineStyleSetLineWidth(3);
LineStyleDrawLine();
# Draw tick lines
LineStyleLineTo(90,length);
while (LineStyleRoll(spacing) <> 1) {
    LineStyleMoveTo(0,0);
    LineStyleLineTo(90,length);
    }
```



More About Pen Position

When you use the LineStyleRoll( ) function, you need to keep in mind that it moves the pointer, but not the pen position. In many scripts this fact is not obvious because most of the local-reference drawing functions automatically move the pen to the origin of the local coordinate system before drawing. In the circle-line script (page 21), for example, the LineStyleDrawCircle( ) function automatically moves the pen to the current pointer position after each LineStyleRoll action. One function that does *not* update the pen position before drawing is the LineStyleLineTo( ) function. This function draws a line from the current pen position to a point specified in the local coordinate system. When you want to use this function in a LineStyleRoll loop to draw a line beginning at the origin of the local coordinate system, as in the tick line script on the preceding page, use the statement "LineStyleMoveTo(0,0)" to move the pen to the current pointer position before drawing.

STEPS
- ☑ reopen the Script Editor window for line styles for the CONTOURS object
- ☑ press the Open icon button and choose Open RVC Object
- ☑ select SINEWAVESCR from the CARTOSMP Project File
- ☑ press [OK] in the Script Editor window and again in the Vector Layer Controls window



The peculiarities of the LineStyleRoll( ) and LineStyleLineTo( ) functions were engineered for a purpose. They make it possible to draw dashed or continuous lines offset from the vector line element. The script in this exercise shows an elegant example, a line symbol that looks like a continuous sine curve. The "curve" is actually made up of small straight-line segments drawn by the LineStyleLineTo( ) function. Each iteration of the "while" loop increments the angle for the sine function by one radian, producing a sinusoidally varying amplitude for the LineStyleLineTo( ) function.

```
# Set line color and width
LineStyleSetColor(0,0,0);
LineStyleSetLineWidth(3);
# Set sine wave parameters
angle = 0;    space = 4;
# Draw line
while (LineStyleRoll(space) <> 1) {
   angle = angle + 1;    # in radians
   a = sin(angle) * 5;   # amplitude
   if (a > 0) then LineStyleLineTo(90,a);
   else LineStyleLineTo(-90,abs(a));
    }
```

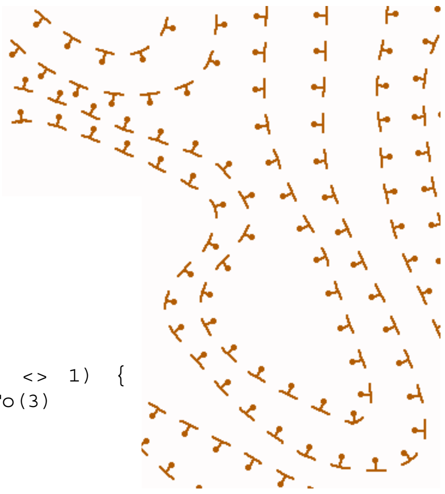
Drawing Dashed Lines

STEPS

- ☑ reopen the Script Editor window for line styles for the CONTOURS object
- ☑ press the Open icon button and choose Open RVC Object
- ☑ select BARBSCR from the CARTOSMP Project File
- ☑ oress [OK] in the Script Editor window and again in the Vector Layer Controls window

To draw simple or complex dashed lines, you can use a "while" loop to alternate LineStyleRoll( ) and LineStyleRollPen( ) actions. The LineStyleRollPen( ) function draws a line along a line element for a specified distance beginning at the current pointer position. The drawing distance is set by the value of the single function parameter. For a simple dashed line with dashes and spaces of equal size, use the same distance for both the LineStyleRoll( ) and LineStyleRollPen( ) functions.

The script in this exercise is a bit more elaborate, adding a tick line with a filled circle at the end in the middle of each dash. In each iteration of the "while" loop, the LineStyleRollPen( ) function draws the first half of a dash, then the tick line and circle are drawn, and finally the second half of the dash is drawn.



```
# Set line color and width
LineStyleSetColor(170,85,0)
LineStyleSetLineWidth(3)

# Set dash parameters
dashSize = 20
half = 0.5 * dashSize

# Set circle parameters
radius = 3;    dofill = 1

# Draw line
while (LineStyleRoll(dashSize) <> 1) {
    dist = LineStyleGetDistanceTo(3)
    if (dist > dashSize) {
        LineStyleRollPen(half)
        LineStyleMoveTo(0,0)
        LineStyleDropAnchor(0)
        LineStyleLineTo(90,half)
        LineStyleDrawCircle(radius,dofill)
        LineStyleMoveToAnchor(0)
        LineStyleRollPen(half)
        }
    else LineStyleRollPen(dist)
    }
```

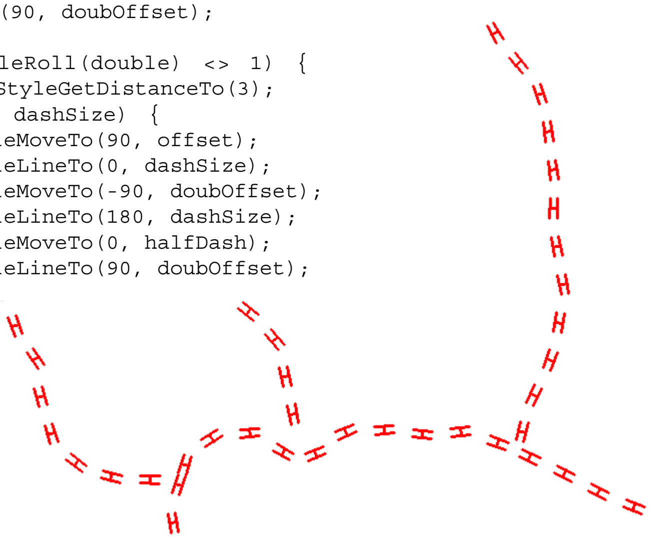
- ☑ when you have completed this exercise, turn off the Show / Hide checkbox for the CONTOURS layer entry in the Display Manager to hide the layer

Double Dashed Lines

The script in this exercise illustrates another dashed line variation. It draws each line with double dashes connected by crossing lines. Each pair of dashes is centered on the line element, which means that the dashes themselves are offset from the line by a distance specified by the *offset* variable. Because no part of the symbol traces the line element itself, the LineStyleLineTo( ) function is used to draw both the dashes and the crossing lines.

```
# Set dash parameters
dashSize = 15;
halfDash = dashSize * 0.5;
double = 2 * dashSize;
offset = dashSize * 0.2;
doubOffset = offset * 2;

# Set line color and width
LineStyleSetColor(255,0,0);
LineStyleSetLineWidth(2);

# Draw double dash line and crossing lines
LineStyleMoveTo(90, offset);
LineStyleLineTo(0, dashSize);
LineStyleMoveTo(-90, doubOffset);
LineStyleLineTo(180, dashSize);
LineStyleMoveTo(0, halfDash);
LineStyleLineTo(90, doubOffset);

while (LineStyleRoll(double) <> 1) {
    dist = LineStyleGetDistanceTo(3);
    if (dist > dashSize) {
        LineStyleMoveTo(90, offset);
        LineStyleLineTo(0, dashSize);
        LineStyleMoveTo(-90, doubOffset);
        LineStyleLineTo(180, dashSize);
        LineStyleMoveTo(0, halfDash);
        LineStyleLineTo(90, doubOffset);
        }
    }
```

STEPS
- ☑ turn on the Show / Hide checkbox for the STREAMS layer in the Display Manager
- ☑ open the Vector Layer Controls window for the STREAMS layer and click on the Lines tab
- ☑ select By Script from the Style option menu and click [Specify...]
- ☑ press the Open icon button in the Script Editor and choose Open RVC Object
- ☑ select DoubDashScr from the CARTOSMP Project File
- ☑ press [OK] in the Script Editor window and again in the Vector Layer Controls window



Handling Multiple Repeat Intervals

STEPS

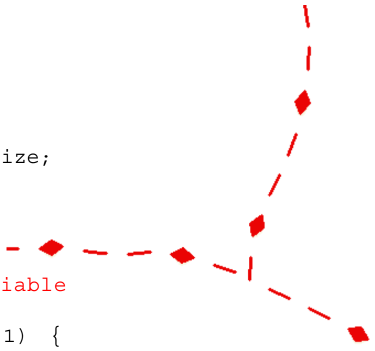
☑ reopen the Script Editor window for line styles for the STREAMS object

☑ press the Open icon button and choose Open RVC Object

☑ select object DASHDISCR from the CARTOSMP Project File

☑ press [OK] in the Script Editor window and again in the Vector Layer Controls window

Line symbols can include components repeated at different intervals along the line, as illustrated by the dashes and diamonds drawn by the script in this exercise. The variable *cumL* in the script keeps track of the cumulative length of dashes and spaces drawn since the last diamond symbol. When the value of *cumL* reaches the spacing distance set for the diamonds, a diamond symbol is drawn instead of a dash, and the value of *cumL* is reset to 0. The diamond symbol is created here using the Line-StyleSideshot( ) function. This function allows you to specify a number of points by angle and distance in the local coordinate system, and connect them to form a polyline by setting the value of the *dodraw* parameter to 1. This script also records the points as a polygon so the diamond shape can be filled.



```
# Set line color and width
LineStyleSetColor(225,0,0);
LineStyleSetLineWidth(2);
# Set dash parameters
dashSize = 12;     half = 0.5 * dashSize;
# Set diamond parameters
spacing = 5 * dashSize;
dodraw = 1;        dofill = 1;
width = 0.3 * dashSize;
cumL = 0; # Cumulative length variable
# Draw line
while (LineStyleRoll(dashSize) <> 1) {
    cumL = cumL + dashSize; # increment cumulative length
    if (cumL >= spacing) {     # draw diamond symbol
        LineStyleRoll(half);
        LineStyleMoveTo(0,0);
        LineStyleRecordPolygon(1);
        LineStyleSideshot(dodraw,0,half,90,width,180,
            half,-90,width);
        LineStyleDrawPolygon(dofill);
        LineStyleRoll(half);
        cumL = 0;      # reset cumulative length to 0
        }
    else {
        LineStyleRollPen(dashSize);# draw dash
        cumL = cumL + dashSize; # increment cumulative length
        }
    }
```

Dashed Thrust Fault Symbol

This CartoScript provides another example of symbols using different repeat intervals. It draws the dashed version of the geologic map symbol for a thrust fault. The triangles are drawn as polygons using anchor points placed at the corners. The bases of the triangle symbols are drawn using the LineStyleRollPen( ) function so that they conform to curves in the line elements. Note that a negative distance can be used with the LineStyleRoll( ) function to move backward along a line element.

STEPS

☑ reopen the Script Editor window for line styles for the STREAMS object

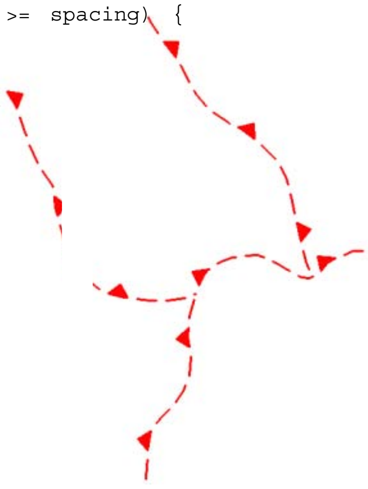
☑ press the Open icon button and choose Open RVC Object

☑ select DASHTHRUSTSCR from the CARTOSMP Project File

☑ press [OK] in the Script Editor window and again in the Vector Layer Controls window

☑ when you have completed this exercise, turn off the Show / Hide checkbox for the STREAMS layer entry in the Display Manager to hide the layer

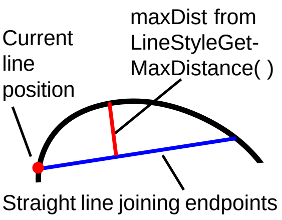
```
# set dash parameters
dashSize = 16;        # length of dashes
halfDash = dashSize * 0.5;
# triangle spacing and dimensions
spacing = dashSize * 6 ;
quartSpace = spacing * 0.25;
triWidth = dashSize;
halfTri = triWidth * 0.5;
height = dashSize * 0.6;
# set line color and width
LineStyleSetColor(255,0,0);
LineStyleSetLineWidth(2);
# initialize variable to control placement of triangles
cumL = spacing;
# draw dashed fault line and triangles
while (LineStyleRoll(halfDash) <> 1) {
   dist = LineStyleGetDistanceTo(3); # distance to end of line
   cumL = cumL + dashSize;
   if (dist > quartSpace and cumL >= spacing) {
      LineStyleDropAnchor(1);
      LineStyleRoll(halfTri);
      LineStyleMoveTo(0,0);
      LineStyleMoveTo(90, height);
      LineStyleDropAnchor(2);
      LineStyleRoll(-halfTri);
      LineStyleRecordPolygon();
      LineStyleRollPen(triWidth);
      LineStyleLineToAnchor(2);
      LineStyleLineToAnchor(1);
      LineStyleDrawPolygon(1);
      cumL = 0;
      }
   else {
      LineStyleRollPen(dashSize);
      cumL = cumL + dashSize;
      }
   }
```



Labeling Contour Lines

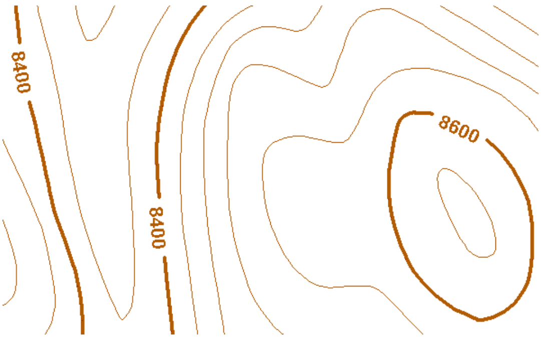
STEPS

- ☑ turn on the Show / Hide checkbox for the CONTOURS layer in the Display Manager
- ☑ open the Vector Layer Controls window for the CONTOURS layer and click on the Lines tab
- ☑ select By Script from the Style option menu and click [Specify...]
- ☑ press the Open icon button in the Script Editor and choose Open RVC Object
- ☑ select CONLABLSCR from the CARTOSMP Project File
- ☑ press [OK] in the ScriptEditor window and again in the Vector Layer Controls window


Current line position
maxDist from LineStyleGet-MaxDistance( )
Straight line joining endpoints

You can design a CartoScript to vary the placement and orientation of line symbol components to accommodate the local direction and shape of the line elements. In this script that draws and labels contours, the elevation labels for the major contours are moved if the local portion of the line is too highly curved, and are inverted if necessary to be readable. (Only the main processing portion of the script is shown on the next page).

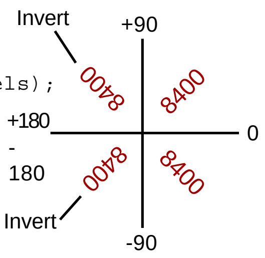
These conditions are checked using the LineStyleGetDirection( ) and LineStyleGetMaxDistance( ) functions. The first parameter of both functions is a value that specifies the length of the portion of the line that you want to examine. The additional parameters of both functions are variables that are assigned values by the function. The LineStyleGetDirection( ) function finds the minimum and maximum direction angles for the specified line segment in the object coordinate reference frame (positive x-axis = 0 degrees). The LineStyleGetMaxDistance( ) function finds the maximum perpendicular distance between the specified portion of the line element and a straight line joining its endpoints (see illustration at left), as well as the direction angle of this straight line (used in this script to orient the contour label). Both functions return a value of 1 if at the end of the line element, or 0 otherwise. The script checks the maximum distance value as well as the change in line direction over the label length to determine if the local portion of the line is too highly curved to place the label there. The direction angle of the label line is used to identify labels that need to be inverted.



- ☑ choose Display / Close when you have completed this exercise

Labeling Contour Lines (continued)

```
if (rem <> 0) {          # draw minor contours
   LineStyleSetLineWidth(width);
   LineStyleDrawLine();
   }
else {                 # draw and label major contours
   LineStyleSetLineWidth(widthBold);
   str$ = sprintf("%d",elev); # read elevation to string variable
   # find length of contour label
   LineStyleTextNextPosition(str$,labelSize,0,1,nextx,
                                       nexty,length);
   begShift = 5 * length;   # offset from beginning of line
   stopLength = begShift; # min label distance from end of line
   spLength = 1.5 * length;     # label length plus spaces

   LineStyleRollPen(begShift); # draw beginning of contour line

   while ((LineStyleGetMaxDistance(spLength,drawAngle,
                                       maxDist)) <> 1) {
      LineStyleGetDirection(0.1 * labelSize,minAngle,maxAngle);
      # find change in line direction over length of label
      devAngle = drawAngle - minAngle;
      # find distance to end of line and compare to stopLength
      remLength = LineStyleGetDistanceTo(3);
      if (remLength < stopLength) break;
      # check deviation distance and angle of line segment
      if ((maxDist < 1.5 * labelSize) and (abs(devAngle) < 15)) {
         LineStyleRoll(0.25*length); # space before drawing label

         # Check if label needs to be inverted to be readable
         isInverse = 0;
         if (abs(drawAngle) > 90) isInverse = 1;
         if (isInverse) {
            LineStyleMoveTo(devAngle, length);
            LineStyleMoveTo(devAngle + 90, halfSize);
            LineStyleDrawText(str$,labelSize,drawAngle+180,1);
            }
         else {
            LineStyleMoveTo(-90,halfSize);
            LineStyleDrawText(str$,labelSize,drawAngle,1);
            }
         LineStyleRoll(1.2 * length);
         LineStyleMoveTo(0,0);
         LineStyleSetLineWidth(widthBold);
         LineStyleRollPen(minDistBetweenLabels);
         }
      else LineStyleRollPen(length);
      }
   # Draw remainder of line
   LineStyleRollPen(remLength);
   }
```



Syncline Line Symbol

STEPS

☑ click on the Add Objects icon button in the Display Manager window

☑ select object SYNCLINE from the CARTOSMP Project File

☑ open the Vector Layer Controls window and click on the Lines tab

☑ select By Script from the Style option menu and click [Specify...]

☑ press the Open icon button in the Script Editor and choose Open RVC Object

☑ select SYNCLINESCR from the CARTOSMP Project File

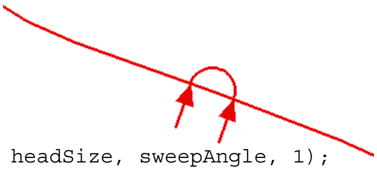
☑ press [OK] in the Script Editor window and again in the Vector Layer Controls window

A CartoScript can be structured to draw different line symbols depending on the attributes attached to the line elements. The script in this exercise draws geological line symbols for the axial trace of a synclinal (downward) fold in layered rocks. If one limb of the fold has rotated beyond a vertical orientation, the fold is overturned, and a special symbol is used. The overturned condition for vector lines in this exercise is indicated by a logical database field.

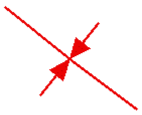
This script draws each line element as a solid line, then places the fold symbol in the middle of the line. The semicircle that forms part of the overturned syncline symbol is drawn using the LineStyleDrawArc( ) function, which draws an arc about a specified center point. The first two function parameters specify an angle and distance to the intended center point of the arc, so you have the option of moving the pen to a new location before drawing the arc. You also specify a starting angle and sweep angle for the arc, and have the option of rotating the entire arc after drawing by assigning a nonzero value for the *rotangle* parameter.

```
#  Read a logical  database  field  (Yes/No)  to  check  if  syncline  is
#  overturned.   Numeric  variable  is  set  to  1  if  yes,  0  if  no
overturned = Syncline.Overturned;
#  dimensions  of  arrow  symbols
arrowSize = 30;    halfSize = arrowSize * 0.5;
headSize = 0.5 * arrowSize;   sweepAngle = 45;
stemSize = arrowSize - headSize * cosd(sweepAngle);
width  =  2;         #  width  of  lines
#  parameters  for  half  circle  in  overturned  syncline  symbol
angle  =  0; shift  =  halfSize;#  angle  and  distance  to  arc  center
radius_x  =  halfSize;         #  radii  of  arc
radius_y = halfSize;
startAngle = -180;     swpAngle = -180;
rotAngle = 0;          isAngleAbs = 0;
#  set  line  color,  width,  and  draw  fold  line
LineStyleSetColor(228,0,0);
LineStyleSetLineWidth(width);
LineStyleDrawLine();
```

Syncline Line Symbol (continued)



```
# draw arrow symbols in middle of each line
LineStyleSetPosition(0.5); # move to middle of line
 if ( overturned == 1 ) {  # draw overturned syncline symbol
    # Draw arrows with 0 line width
    LineStyleRoll(-halfSize);
    LineStyleDropAnchor(1);
    LineStyleSetLineWidth(0);
    LineStyleMoveTo(-90, arrowSize);
    LineStyleDropAnchor(2);
    LineStyleDrawArrow(90, arrowSize, headSize, sweepAngle, 1);
    LineStyleMoveTo(0, arrowSize);
    LineStyleMoveTo(-90, arrowSize);
    LineStyleDropAnchor(3);
    LineStyleDrawArrow(90, arrowSize, headSize, sweepAngle, 1);

    # Draw arrow stems and arc with linewidth = width
    LineStyleSetLineWidth(width);
    LineStyleMoveToAnchor(2);
    LineStyleLineTo(90, stemSize);
    LineStyleMoveToAnchor(1);
    LineStyleDrawArc(angle, shift, radius_x, radius_y,
                            startAngle, swpAngle, isAngleAbs);
    LineStyleMoveToAnchor(3);
    LineStyleLineTo(90, stemSize);
    }
else {    # draw upright syncline symbol
    # draw arrows with 0 line width
    LineStyleDropAnchor(1);
    LineStyleSetLineWidth(0);
    LineStyleMoveTo(90, arrowSize);
    LineStyleDropAnchor(2);
    LineStyleDrawArrow(-90, arrowSize, headSize, sweepAngle, 1);
    LineStyleMoveToAnchor(1);
    LineStyleMoveTo(-90, arrowSize);
    LineStyleDropAnchor(3);
    LineStyleDrawArrow(90, arrowSize, headSize, sweepAngle, 1);

    # redraw arrow stems
    LineStyleMoveToAnchor(2);
    LineStyleSetLineWidth(width);
    LineStyleLineTo(-90, stemSize);
    LineStyleMoveToAnchor(3);
    LineStyleLineTo(90, stemSize);
    }
```



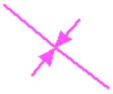
This script draws the dip arrows for the overturned syncline symbol on the right side of a line element. Line directions have been set in the vector object to accomodate this script design.

Add Elements to LegendView

STEPS

☑ press the Select icon button on the View window

☑ in the View window, left-click on one of the two lines drawn with the upright syncline symbol



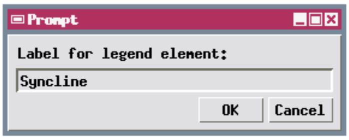
☑ in the Display Manager, right-click on the Syncline layer entry and select Add Active Line to Legend from the popup menu

☑ when the Prompt window for the legend element label appears, enter “Syncline” in the text field and press [OK]

Vector style samples appear automatically in LegendView when you style points, lines, or polygons using the All Same, By Attibute, or By Theme style options. But when you style vector elements using a script, the “styles” created by your script do not automatically appear in LegendView; instead the current All Same style for the element appears in LegendView by default.

You can create LegendView samples for vector elements styled by script by selecting from the View window a representative element for each drawing “style” and adding it to LegendView.

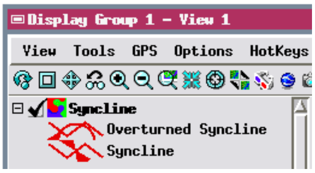
In this exercise you select two lines from the syncline vector layer, one styled with the normal upright syncline symbol and the other with the overturned syncline symbol. When each LegendView sample is rendered in the window, the script is executed using the database attributes of the element you selected for it.



Prompt

Label for legend element:

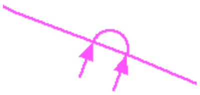
Syncline



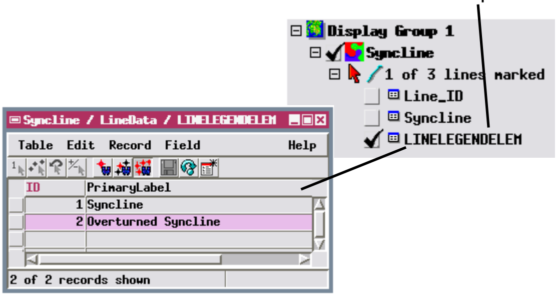
Display Group 1 - View 1

Syncline
Overturned Syncline
Syncline

☑ repeat the last three steps for the line drawn with the overturned syncline symbol, entering “Overturned Syncline” for the label



A special database table is created to identify the elements you have selected for the legend. The table also stores the label text for each sample.



Display Group 1
Syncline
1 of 3 lines marked
Line_ID
Syncline
✓ LINELEGENDELEM

Syncline / LineData / LINELEGENDELEM

ID	PrimaryLabel
1	Syncline
2	Overturned Syncline

2 of 2 records shown

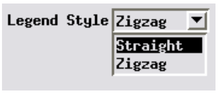
Script Variables for Legend Samples

Notice that the syncline symbols drawn in LegendView in the previous exercise overlap each other because they appear too large for the legend spacing. LegendView assigns a rectangular area to each sample, and the general scaling parameters included in your script for map rendering may not provide a good fit to that area.

Several script variables let you set up special processing to better scale symbols for legends. When the Display process renders a sample in LegendView using a script, the variable *DrawingLegendView* is automatically assigned a value of 1. (Its value is 0 when the script is rendering elements in the view.) You can therefore set up a conditional loop in the script that checks this value and executes only when the script is drawing a sample for LegendView.

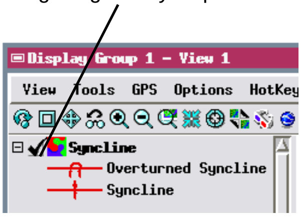
When a legend sample is being rendered, class *SampleRect* is created automatically to allow you to access the coordinates and size of the sample rectangle. *SampleRect* is an instance of the general class *Rect*, which includes methods GetHeight() and GetWidth() that you can use to find either dimension of the rectangle. The CartoScript in this exercise uses the height of the sample rectangle to scale the syncline symbols in LegendView. The relevant portion of the script is excerpted below.

STEPS
- ☑ open the Vector Layer Controls window and click on the Lines tab
- ☑ select Straight from the Legend Style menu on the lower part of the panel



- ☑ reopen the Script Editor window for line styles
- ☑ press the Open icon button and choose Open RVC Object
- ☑ select object SynScrLgn from the CARTOSMP Project File
- ☑ press [OK] in the Script Editor window and again in the Vector Layer Controls window

The default shape for a line sample in LegendView is zigzag. Complex line symbols drawn by CartoScript look better in LegendView when drawn with the Straight legend style option.



```
# These variables control the length of the arrows
# ArrowLengthMap is the desired arrow length in mm,
# assuming vector coordinates are in meters:
if (DrawingLegendView == 1) {
    arrowSize = 0.5 * SampleRect.GetHeight();
}
else {
    arrowLengthMap = 4;
    arrowSize = arrowLengthMap * scale / 1000;
}
```

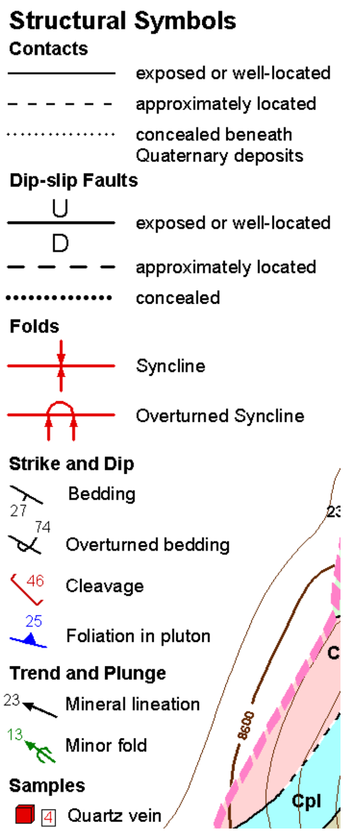
- ☑ close the current display group when you have completed this exercise

CartoScript Samples in Printed Legends

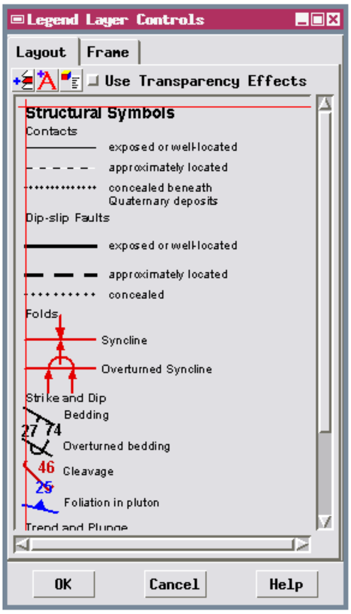
STEPS
- ☑ choose Display / Open from the Display Manager
- ☑ select PRINTLAYOUT from the GGMAP Project File
- ☑ in the Display Manager, click on the Legend icon button for the Symbols legend

- ☑ examine the Legend Layer Controls window, then click [OK]

Once you have selected legend samples for vector elements that are being rendered by a script, you can also create a legend with these symbols in a map layout for printing. Simply create a multi-object legend, use the Legend Layer Control window to add the appropriate element type (point, line, or polygon) from the vector object, and the script-styled legend samples and their labels appear automatically in the legend. The Legend Layer Controls window allows you to adjust the positions of legend entries and to change the label text and style if desired. The tutorial booklet *Making Map Layouts* includes several exercises to guide you through the creation of a multi-object legend.



The print layout you open in this exercise includes a multi-object legend of geologic line and point symbols rendered by CartoScript. All of the script objects used are also included in the Project File.



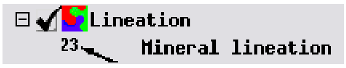
Structural Symbols
Contacts
exposed or well-located
approximately located
concealed beneath Quaternary deposits
Dip-slip Faults
exposed or well-located
approximately located
concealed
Folds
Syncline
Overturned Syncline
Strike and Dip
Bedding
27 74
Overturned bedding
46 Cleavage
25
Foliation in pluton
Trend and Plunge

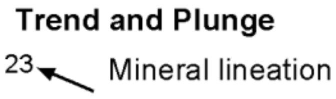
Scripting Legend Sample Position

Point and line symbols should be rendered in a printed legend at the same scale as they are on the map. If you have scaled your symbols to a map scale, they will appear at the correct scale automatically in a multi-object legend. But there are instances in which you might want to have the script adjust the position of samples both in LegendView and in a multi-object legend. The script variable *DrawingSample* is automatically set to 1 whenever the script renders either type of legend sample, allowing you to set up instructions in the script to reposition symbols in the legend sample.

The script in this exercise is a version of one shown in the earlier exercise Orienting Symbols by Attribute. It draws labeled arrows that in this case show the trend and plunge of mineral lineations at various map locations. In the map the base of the arrow is at the outcrop location. The drawing instructions are modified when a legend sample is being rendered to center the arrow on the point and move it down a few pixels to better align vertically with the legend label. The relevant portion of the script is excerpted below.

STEPS
- ☑ in the Display Manager, open the Vector Layer Controls window for the Lineation vector layer in Group 1
- ☑ click on the Points panel in the Vector Layer Controls window and open theScript Editor window for point styles
- ☑ note how legend variables are used in the script
- ☑ click [OK] in the Script Editor window and again in the Vector Layer Controls window





```
# Draw arrow with zero line width, tip of stem at data point
# but centered on point if legend sample
if (DrawingSample == 1) {
    LineStyleMoveTo(oppTrend, arrowLength / 2);
    LineStyleMoveTo(-90,5);
}

LineStyleDropAnchor(0);             # anchor at data point
LineStyleSetLineWidth(0);
LineStyleDrawArrow(direction, arrowLength, headSize, angle, 1);
LineStyleDropAnchor(1);             # anchor at tip of arrow

# Draw arrow stem with desired line width
LineStyleMoveToAnchor(0);
LineStyleSetLineWidth(lineWidth);
LineStyleLineTo(Direction,stemLength);
LineStyleMoveToAnchor(1);
```

CartoScript Function List

Style Control Functions

`LineStyleSetCapJoinType`	set square or rounded ends for lines and polyline segments
`LineStyleSetColor`	set RGB values (0-255) for line color
`LineSyleSetCoordType`	set coordinate type for input values (object or millimeters)
`LineStyleSetFont`	set name of font to use for text
`LineStyleSetLineWidth`	set width of lines to be drawn
`LineStyleSetScale`	set scale factor applied to input distances
`LineStyleSetTextColor`	set RGB values (0-255) for text color

Label Optimization

`LineStyleAddToOptimizer`	access built-in label placement optimizer

Functions that reference position within input line elements

Navigation

`LineStyleNextVertex`	move pen to next vertex along line element
`LineStylePrevVertex`	move pen to previous vertex along line
`LineStyleRoll`	move pointer a specified distance along line element
`LineStyleSetPosition`	move pen to specified relative position along line element (0.0 = start, 1.0 = end)

Drawing

`LineStyleRollPen`	move pen to pointer position and draw line for specified distance along line element

Information

`LineStyleGetDirection`	find minimum and maximum direction angles of line element segment
`LineStyleGetDistanceTo`	find distance from pointer position to next vertex (1), previous vertex (2), end (3), or start (4) of line element
`LineStyleGetLineCurvature`	find curvature of line element segment
`LineStyleGetMaxDistance`	find maximum perpendicular distance from line segment to straight line connecting segment start and end
`LineStyleGetPosition`	find current relative position of pointer along line element (0.0 = start, 1.0 = end)
`LineStyleIsClosed`	test if current line element forms a closed polygon (1) or not (0)

Note: the line segment operated on by certain functions in the last group above begins at the current pointer position along the line element and extends to a specified distance from it.

CartoScript Function List (continued)

Functions that reference the current local coordinate system

Navigation

`LineStyleDropAnchor`	record current pen position as anchor number for later use
`LineStyleMoveTo`	move pen to location specified by direction and distance from current pen location
`LineStyleMoveToAnchor`	move pen to specified anchor location

Drawing

`LineStyleDrawArc`	draw arc
`LineStyleDrawArrow`	draw arrow to location specified by direction and distance from current pen location
`LineStyleDrawCircle`	draw filled or unfilled circle
`LineStyleDrawCone`	draw cone
`LineStyleDrawCube`	draw cube
`LineStyleDrawCylinder`	draw cylinder
`LineStyleDrawEllipse`	draw filled or unfilled ellipse
`LineStyleDrawPolygon`	draw polygon connecting previously recorded points
`LineStyleDrawPolyline`	draw polyline connecting previously recorded points
`LineStyleDrawRectangle`	draw filled or unfilled rectangle
`LineStyleDrawText`	draw text label with specified text string
`LineStyleDrawTextBox`	draw boxed text label
`LineStyleDrawThreePointArc`	draw arc connecting three points
`LineStyleLineTo`	draw line to location specified by direction and distance from current pen location
`LineStyleLineToAnchor`	draw line from current pen position to specified anchor location
`LineStyleRecordPolygon`	start or stop recording polygon vertex locations
`LineStyleSideshot`	specify sequence of positions by direction and distance from current pen position and optionally connect to form polyline
`LineStyleTextNextPosition`	compute length and end position of text string

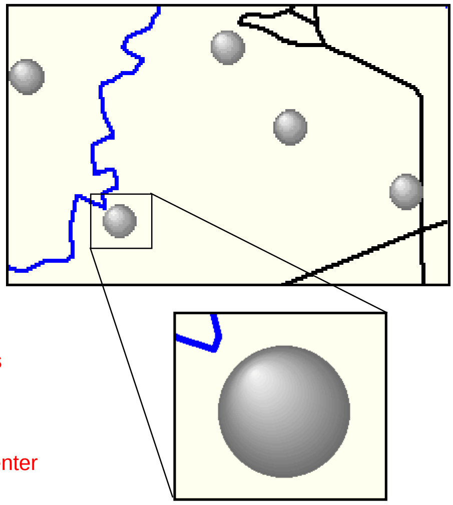
Functions that draw or transform entire input line elements

`LineStyleDrawLine`	draw entire vector or CAD line element
`LineStyleRestoreLine`	restore original line coordinates
`LineStyleSpline`	replace line element with splined line
`LineStyleThinLine`	replace line element with thinned line

Points: Shaded Sphere

Generates a shaded sphere. This script is data-independent and suitable for use with pinmaps or vectors.

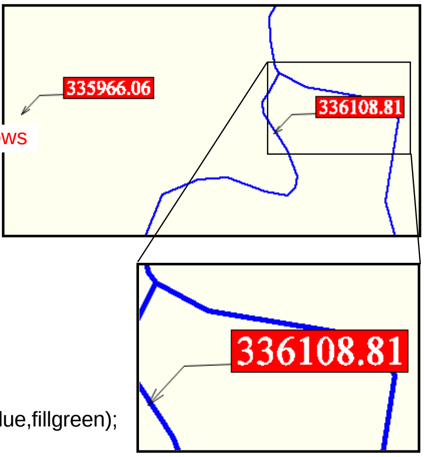
```
#Draws a shaded sphere by drawing
#a series of ofset circles of progressively
# smaller radius and brighter white
radius = 10;
maxc = 255;
minc = 110;
stepnum = 31;
angle = 125;
p = 0.76;  # overlap variable
k = (maxc - minc) / (stepnum + 1);
      # k is the shading rate
delta = radius * p / stepnum;
rd = radius / stepnum;
for i = 0 to stepnum step 1 {
   color = minc + i * k; #increase brightness
   LineStyleSetColor(color,color,color);
   LineStyleDrawCircle(radius,1);
   radius = radius - rd;  #reduce radius
   LineStyleMoveTo(angle,delta);    #shift center
   }
```



Points: Coordinate Labels

Generates boxed text (X coordinate) attached to point's position by arrow. This script is data-independent. For vector objects only unless modified.

```
# filled text box with bent and arrowed line
# pointing at the point
fillred = 255;
fillblue = 0;
fillgreen = 0;
LineStyleSetColor(0,0,0); #black lines + arrows
LineStyleDropAnchor(0);
#Draw arrow head
LineStyleLineTo(60,5);
LineStyleMoveToAnchor(0);
LineStyleLineTo(30,5);
LineStyleMoveToAnchor(0);
# Draw bent line
LineStyleLineTo(45,15);
LineStyleLineTo(0,15);
# Draw filled box
LineStyleSetTextColor(255,255,255,fillred,fillblue,fillgreen);
str$ = sprintf("%5.2f",Internal.x);
LineStyleDrawTextBox(str$,10,0,2,1);
return 1;
```

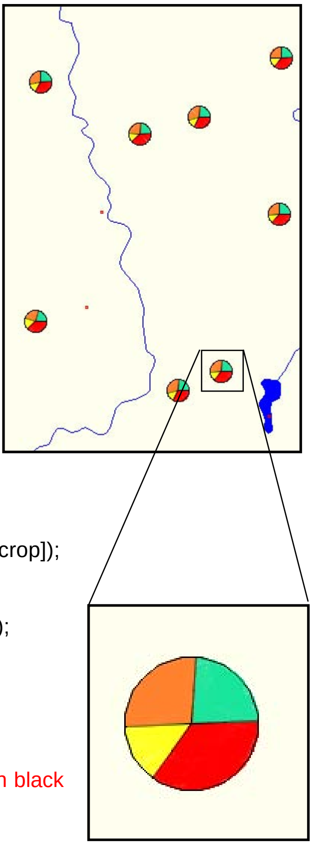


Variation: change color of text or fill

Points: Pie Graph

Generates a mini pie chart for each point in the data. This script is data-dependent; change yields to values appropriate for your data.

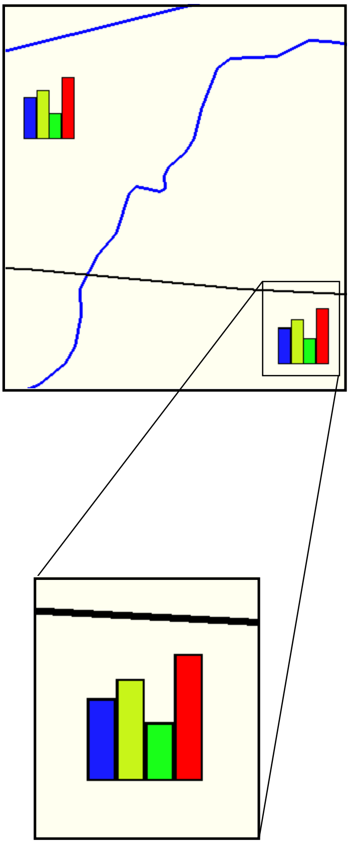
```
numofcrops = 4;           PieRadius = 10;
array numeric PartSize[4];   array numeric Value[4];
array numeric red[4];    array numeric green[4];    array numeric blue[4];

Value[1] = yield.wheat ;
red[1] = 25;   green[1] = 228;   blue[1] = 155;
Value[2] = yield.oats;
red[2] = 255;   green[2] = 128;   blue[2] = 45;
Value[3] = yield.corn;
red[3] = 255;   green[3] = 255;   blue[3] = 25;
Value[4] = yield.sorghum;
red[4] = 255;   green[4] = 0;   blue[4] = 0;

#find sum of values
SumValue =0;
for i = 1 to numofcrops step 1 {
    SumValue = Value[i] + SumValue;
}
#calculate partsize for each crop
if (SumValue != 0) {
    for i = 1 to numofcrops step 1 {
        PartSize[i]= Value[i] *360 / SumValue;
        }
    #draw pie piece fill areas
    LineStyleDropAnchor(0);  # center of symbol
    present = 0;
    for crop = 1 to numofcrops step 1  {
        next = present + PartSize[crop];
        LineStyleSetColor(red[crop],green[crop],blue[crop]);
        if (PartSize[crop] >0) {
            for angle = present to next -1 step 1 {
                    LineStyleLineTo(angle,PieRadius);
                    LineStyleMoveToAnchor(0);
                    }
            present = next;
            }
        }
    LineStyleSetColor(0,0,0);   #draw the edge lines in black
    angle = 0;
    for crop = 1 to numofcrops step 1 {
        angle = angle + PartSize[crop];
        LineStyleLineTo(angle,PieRadius);
        LineStyleMoveToAnchor(0);
        }
    LineStyleDrawCircle(PieRadius); #draw outer circle
    }
```



Points: Bar Graph

Generates data-dependent bar graph with color that varies from crop to crop. Change yields to values appropriate for your data.

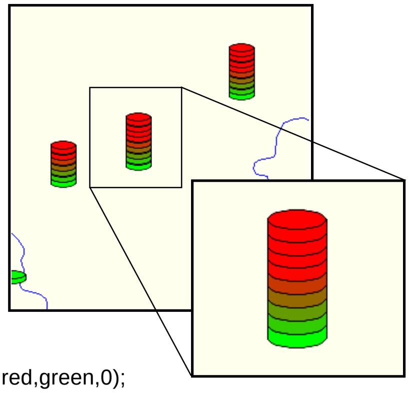
```
#Bar Graph of Crop Yield at each point
LineStyleDropAnchor(0);  # center of symbol

# display color of bar is set for each of the crops
numofcrops = 4;
array BarLength[4],red[4],green[4],blue[4];
BarLength[1] = yield.wheat;
red[1] = 25; green[1] = 28; blue[1] = 255;  # cyan
BarLength[2] = yield.oats;
red[2] = 200; green[2] = 245; blue[2] = 25; # yellow
BarLength[3] = yield.corn;
red[3] = 25; green[3] = 255; blue[3] = 25; # green
BarLength[4] = yield.sorghum;
red[4] = 255; green[4] = 0; blue[4] = 0; # red

#draw vertical bars
thickness =4;
underline = thickness + 2;
LineStyleSetCapJoinType(1,1);

for crop = 1 to numofcrops step 1  {
    if (BarLength[crop]>0) {
       LineStyleMoveTo(0,(crop -1)*underline);
                              #move bottom to the right

      #draw black line first
      LineStyleSetLineWidth(underline);
      LineStyleSetColor(0,0,0);
      LineStyleLineTo(90,BarLength[crop]+2);

      #restore start position for colored line
      LineStyleMoveToAnchor(0);
      LineStyleMoveTo(0,(crop -1)*underline);
      LineStyleMoveTo(90,1);

      #draw colored line over black
      LineStyleSetLineWidth(thickness);

LineStyleSetColor(red[crop],green[crop],blue[crop]);
      LineStyleLineTo(90,BarLength[crop]);
            LineStyleMoveToAnchor(0);
      }
   }
```



Points: Round Stack

Generates data-dependent round stacks with color that varies with the height of the stack. Change yield.oats to a value appropriate for your data.

```
# Oat yield is displayed by using a stack
# of cylinders; color of each cylinder varies
# as the height changes with red increasing
# and green decreasing at proportionate
# rates

LineStyleSetColor(0,0,0);
height = 5;
depth = 8;
width = 22;
colorate = 10;
for i = 0 to yield.oats step 5 {
    red = i*colorate;  # red increases
    green = 255 - red;  # green decreases
    LineStyleDrawCylinder(width,depth,height,red,green,0);
    LineStyleMoveTo(90,height);
    }
```

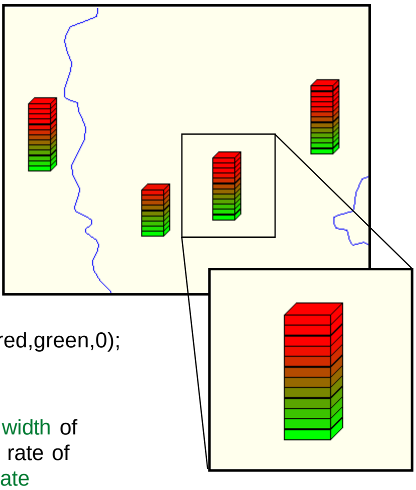


Variation: change height, depth, width of each stack component or change rate of color change by altering the colorate

Points: Square Stack

Generates data-dependent cubic stacks with color that varies with the height of the stack. Change yield.wheat to a value appropriate for your data.

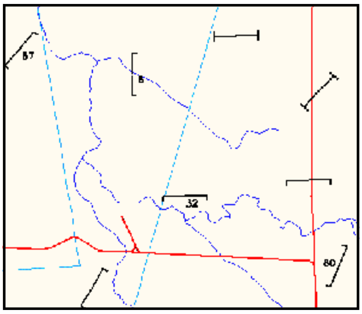
```
# Wheat yield is displayed by using a stack
# of cubes; color of each cube varies as the
# height changes with red increasing and
# green decreasing
LineStyleSetColor(0,0,0);
height = 5;
depth = 8;
width = 22;
colorrate = 10;
for i = 0 to yield.wheat step 5 {
      red = i*colorrate;  # red increases
      green = 255 - red;  # green decreases
      LineStyleDrawCube(width,depth,height,red,green,0);
      LineStyleMoveTo(90,height);
      }
```



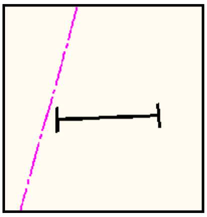
Variation: change height, depth, width of each stack component or change rate of color change by altering the colorate

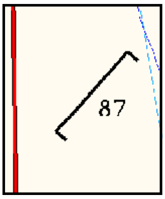
Points: Geologic Cleavage

Implements the standard geological cleavage strike and dip symbol with vertical dip variant. Change cleavage.strike and cleavage.dip to values appropriate for your data.



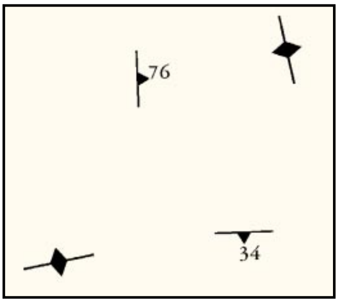
```
LineStyleDropAnchor(0); # center of symbol
LineStyleSetColor(0,0,0); # black symbols
LineStyleSetCapJoinType(1,1); # squared off lines
strike1 = cleavage.strike;
dip1 = cleavage.dip;

scaling = 7;
radius = 4*scaling;
LineStyleSetLineWidth(scaling);
textheight = 6 *scaling;

# Draw Strike Line
LineStyleLineTo(strike1,16*scaling);
LineStyleDropAnchor(1);
LineStyleMoveToAnchor(0);
LineStyleLineTo(strike1 - 180,16*scaling);

if (dip1==90){ # use two end lines to mark vertical bedding
    LineStyleMoveTo(markangle,4*scaling);
    LineStyleLineTo(markangle - 180, 8*scaling);
    LineStyleMoveToAnchor(1);
    LineStyleMoveTo(markangle,4*scaling);
    LineStyleLineTo(markangle - 180, 8*scaling);
    }

else { # dip direction line
    # rotate symbol by 180 when dip direction indicates
    if (bedding.dip_direction) rot = 180 else rot = 0;
    markangle = strike1-90+rot;
    if (markangle > 360) markangle = markangle -360;
    if (markangle < 0)  markangle = markangle + 360;

    LineStyleLineTo(markangle,4*scaling);
    LineStyleMoveToAnchor(1);
    LineStyleLineTo(markangle,4*scaling);
    }

if ((dip1<90) and (dip1>0)) { # label non-horizontal and non-vertical points
    LineStyleMoveToAnchor(0);
    LineStyleMoveTo(markangle,2*scaling);
    LineStyleSetTextColor(0,0,0);
    str$ = sprintf("%2d",dip1);

    if ((markangle>90) and (markangle<270)) {
        LineStyleTextNextPosition(str$,textheight,0,0,nextx,nexty,tlength);
        LineStyleMoveTo(180,tlength);
        }

    if (markangle>180) {
        LineStyleMoveTo(- 90,textheight);
        }

    LineStyleDrawText(str$,textheight,0,2);
    }
```

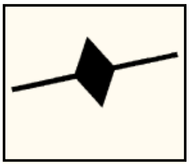


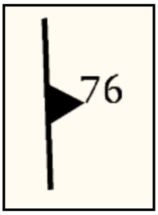


Points: Geologic Foliation Dip and Strike

Generates a foliation symbol based on dip, strike and dip direction. This script is data-dependent; change foliation.strike, foliation.dip and foliation.dip_direction to values appropriate for your data.





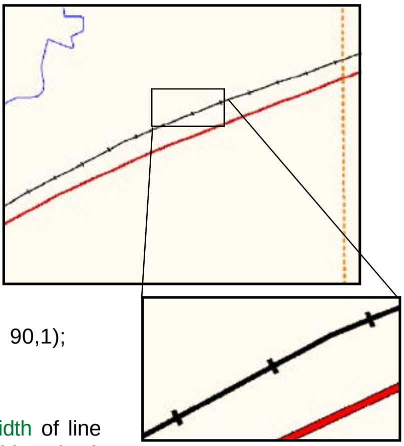


```
LineStyleDropAnchor(0); # center of symbol
LineStyleSetColor(0,0,0); # black symbols
LineStyleSetCapJoinType(1,1); # squared off
strike1 = foliation.strike;
dip1 = foliation.dip;
scaling = 7;
radius = 4*scaling;
LineStyleSetLineWidth(scaling);
textheight = 6 *scaling;

# Draw Strike Line
if (dip1!=0) {
      LineStyleLineTo(strike1,16*scaling);
      LineStyleMoveToAnchor(0);
      LineStyleLineTo(strike1 - 180,16*scaling);
      LineStyleMoveToAnchor(0);
      }

# rotate symbol by 180 when dip direction indicates
if (foliation.dip_direction) then rot = 180
      else rot = 0;
strikeangle =strike1+rot;
if (strikeangle > 360) then strikeangle = strikeangle -360;
markangle = strikeangle -90;
if (markangle < 0) then markangle = markangle + 360;
sidesize=8*scaling;

# design the dip direction triangle
LineStyleMoveToAnchor(0);
LineStyleMoveTo (strike1 + rot-180,sidesize/2); # starting position
LineStyleRecordPolygon(1);
LineStyleMoveTo(strikeangle-60,sidesize);
LineStyleMoveTo(strikeangle+60 ,sidesize);
if (dip1==90) {  # extend triangle into a diamond for vertical foliation
      LineStyleMoveTo(strikeangle+120,sidesize);
      LineStyleMoveTo(strikeangle-120,sidesize);
      }
      else LineStyleMoveTo(strikeangle-180,sidesize);
LineStyleRecordPolygon(0);
LineStyleDrawPolygon(1);  # draw triangle or diamond

if (dip1<90) { # label non-vertical foliations
      LineStyleMoveToAnchor(0);
      LineStyleMoveTo(markangle,sidesize/1.4);
      LineStyleSetTextColor(0,0,0);
      str$ = sprintf("%2d",dip1);
      if ((markangle>90) and (markangle<270)) {
         LineStyleTextNextPosition(str$,textheight,0,0,nextx,nexty,tlength);
         LineStyleMoveTo(180,tlength);
         }
      if (markangle>180) {
         LineStyleMoveTo(- 90,textheight);
         }
      LineStyleDrawText(str$,textheight,0,2);
      }
```

Lines: Railroad

Uses periodic thin rectangles at ninety degrees to the line to create a classic railroad line style.

```
LineStyleSetColor(0,0,0);
# thickness of line and cross bars
width = 5;
# distance between each
period = 60;
# length of crossbar
crossbar = 15;

# draw cross pieces
LineStyleSetLineWidth(width,0);
while (LineStyleRoll(period)!=1) {
     LineStyleDrawRectangle(crossbar,width, 90,1);
     }

# draw the line
LineStyleDrawLine();
```

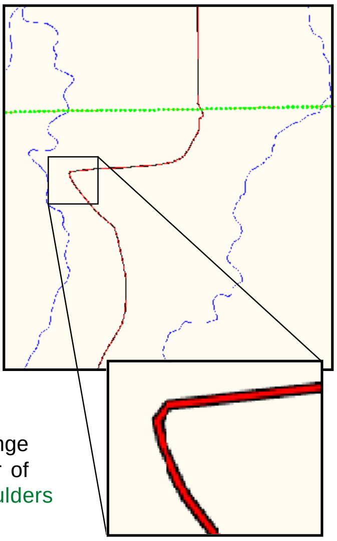


Variation: width of line or period and length of the crossbar

Lines: Highways

Generates red highway lines with black borders.

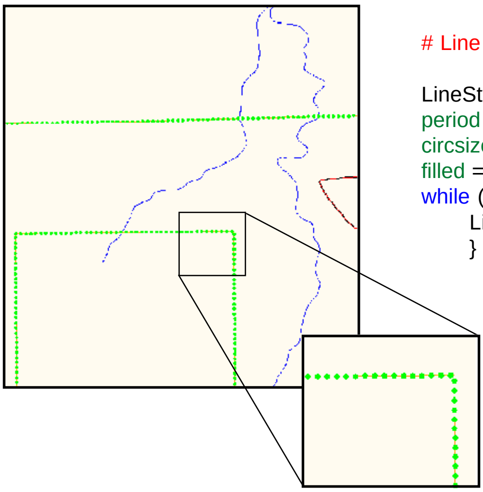
```
# Two overlapping lines: thinner red line
#  over thicker black line with intersections
# cleaned up by shortening the black

rdshoulder = 1;   #  edit these values to
rdwidth = 4;       # change thicknesses
if  (rdshoulder !=0 )   {
     totalwidth = rdwidth + 2* rdshoulder;
     halfrd = totalwidth /2;
     LineStyleSetColor (0,0,0);
     # begin drawing half a road width in
     LineStyleRoll(halfrd);
     # stop half a road width before end
     nearend = LineStyleGetDistance(3) - halfrd;
     # draw black line
     LineStyleRollPen(nearend);
     }
# now draw red line over
LineStyleSetColor(255,0,0);
LineStyleSetLineWidth(rdwidth);
LineStyleDrawLine();
```



Variation: change thickness. color of rdwidth, rdshoulders

Lines: Dotted

Uses periodic circles to make a dotted line style.



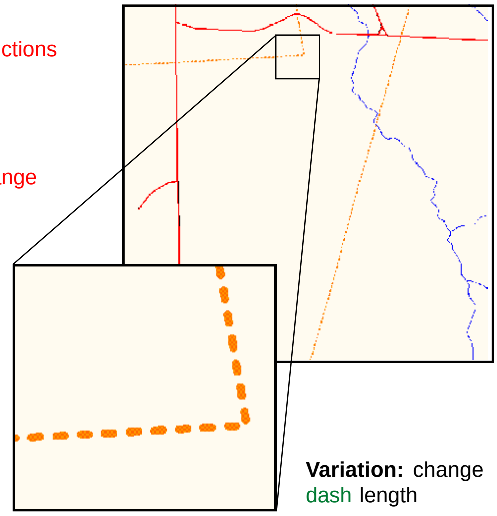
```
# Line of solid green circles

LineStyleSetColor(0,255,0);
period = 25;  # distance between dots
circsize = 8;
filled = 1;  # solid circles as dots
while (LineStyleRoll(period) !=1) {
      LineStyleDrawCircle(circsize,filled);
      }
```

Variation: change filled to 0 for small circles instead of dots

Lines: Even Dashes

Generates equal spaced orange dashed line.

```
# CartoScript : Even Dashes
# use alternating roll and roll pen functions
# over the same dash distance

dash = 8 ;

LineStyleSetColor(255,128,0);  # orange
LineStyleSetLineWidth(8,0);

while (LineStyleRoll(dash) != 1) {
     LineStyleRollPen(dash);
}
```



Variation: change dash length

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 professionals with small projects. You can download TNTmips Free from MicroImages' web site.

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

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

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

Index

MicroImages, Inc.

www.microimages.com