

VIS Shape Path Functions

1 December 2025

Copyright © **Victor Vella** (2025)
All rights reserved.



Other Related ETAC Documents

ETAC_Preliminaries.pdf	Preliminaries before using ETAC
ETACOverview.pdf	An Overview of ETAC
ETACProgLang(Official).pdf	The Official ETAC Programming Language
ETACVisualInteractionSystem.pdf	ETAC: Visual Interaction System
RunETAC.chm	Run ETAC Scripts Help
ETACWithCPP.pdf	ETAC: Interacting with C++
ETACCompiler.pdf	The ETAC Compiler
ETACCompiler.chm	ETAC Compiler Help
ETACErrorCodes.pdf	ETAC Compilation and Run-time Error Codes
FunctionsETACScriptLib.pdf	Functions ETAC Script Library
SplineScriptFnts.pdf	Spline ETAC Script Library
PathScriptFnts.pdf	Path ETAC Script Library
VISPointSetFnts.pdf	VIS Point Set Functions

Legal Information

ETAC and  (the ETAC logo) are unregistered trademarks (™) of Victor Vella for *computer software incorporating an implementation of a computer programming language*. There may be other owners of the “ETAC” trademark used for other purposes.

This document is copyright © by Victor Vella (2025). All rights reserved. Permission is hereby granted to make any number of exact electronic copies of this document without any remuneration whatsoever. Permission is also granted to make annotated electronic copies of this document for personal use only. Except for the permissions granted, and apart from any fair dealing as permitted under the relevant Copyright Act, no part of this document may be reproduced or transmitted in any form or by any means without the express permission of the author. The copyright of this document shall remain entirely with the original copyright holder.

The author of this document shall not be liable for any direct or indirect consequences arising with respect to the use of all or any part of the information in this document, even if such information is inaccurate or in error. The information in this document is subject to change without notice.

Contents

Contents.....	iii
Document Conventions.....	iv
1. Introduction.....	1
2. SPF Context.....	1
3. SPF Commands.....	2
3.1 Command Summary.....	2
3.2 Command Definitions.....	3
4. SPF Functions.....	7
4.1 Function Summary.....	7
4.2 Function Definitions.....	8
4.2.1 Shape Path Creation Functions.....	8
4.2.2 Current Path Modification Functions.....	30
4.2.3 Current Path Transformation Functions.....	36
5. Creating Custom Arrowheads.....	39
5.1 The Procedure of Positioning an Arrowhead.....	39
5.2 Examples of Creating an Arrowhead.....	41
5.2.1 Path with a Standard Arrowhead Attached.....	41
5.2.2 Path with Two Arrowheads Attached.....	42
5.2.3 Path with Two Separate Arrowheads.....	43
Bibliography.....	46
Glossary.....	47

Document Conventions

The following symbolic conventions are used in this document.

Symbol	Meaning
$\langle x \rangle$	separates x as a unit of information from the surrounding text.
$x \cdots$	middle ellipsis means zero, one, or more of the same kind as x .
$[x]$	means that x optional.
(x)	groups x as a unit.
$\dots$	ellipsis represents omitted text (as usual).
<i>text</i>	maroon coloured italic text is a link to the text's definition.
<u>text</u>	underlined green text is a link into the document.
text	bold green text is a link into the document.
♦	indicates the end of a block of text.

VIS Shape Path Functions

This document is for version **1-0-4-ena** of the internal **VIS Shape Path Functions** library for version **1-1** of the **ETAC™ Programming Language** implemented in RunETAC.exe and AppETAC.dll version **4-0-6-ena**. The source file of the library is in VISShapePathFnts.etac.

(Australian English)

1. Introduction

ETAC™ (pronounced: E-tack) is a syntactically simple but extremely versatile dictionary and stack based interpreted script programming language. The ETAC programming language is not based on any other programming language. Familiarity with the ETAC programming language is required to fully understand this document. See the document ETACOverview.pdf for an overview of the language, and also the document ETACProgLang(Official).pdf for the official definition of the language.

The VIS Shape Path Functions library is included in the VIS (Visual Interaction System) version of the ETAC interpreter, and only requires the inclusion of the VISGlobal.PTAC file to use in ETAC programs.

An ETAC program can specify the points in the point sequence of a curve (or line) item component instance explicitly, or typically, by using a combination of functions from a set of (currently) 42 functions called ‘SPF functions’. “SPF” stands for **Shape Path Functions**. (We will ignore the fact that the term *SPF functions* translates directly to “Shape Path Functions functions”.) *SPF functions* are mainly used with curve items because they produce the points of a sequence of Bezier curves, but can also be used with line items with the proper manipulations.

A particular data object context containing standard parameters is required for use by the SPF functions. The particular data object is called an “*SPF context*”, and is always used with SPF functions. Various SPF functions use appropriate parameters in the *SPF context*, and possibly passed arguments in addition. The parameters have different meanings depending on the SPF function using them.

An *SPF context* is created with default parameter values. There 21 parameters beginning with the prefix **<@spf>**. All the SPF functions use the *current path* (sequence of points) within the *SPF context*.

2. SPF Context

All SPF functions require an *SPF context* which is created implicitly via various SPF commands, especially the **@SPF** command. An *SPF context* contains parameters used by the SPF functions. The meaning of the parameter values depends on the SPF function that uses the relevant parameters. Being an ETAC data object, an *SPF context* is automatically pushed onto the dictionary stack only during the execution of the SPF functions within a specified procedure.

The table below summarises the parameters contained in an *SPF context* along with their default values.

SPF Context Parameters

Parameter	Description
@spfPath :- []	Contains the <i>current path</i> used by all SPF functions within the <i>SPF context</i> . The value of this path is returned by the @SPF command and typically assigned to @ciPoints of a VIS curve item component instance.
@spfAncPoint :- 0	Used with custom arrowheads to anchor the end of the <i>current path</i> to the specified position on the arrowhead.
@spfApexRadius :- 0.0	Contains the radius of curved apices of some shapes. Some shapes have vertices that can be curved by the specified radius.
@spfAppend :- true	Determines whether a path is to be connected to the beginning or end of the <i>current path</i> .
@spfArrowBase :- 6	Used with custom arrowheads to move the arrowhead to the specified position relative to the end of the <i>current path</i> .
@spfClose :- false	Indicates whether the end point of a path is to be connected to the start point by a straight line.
@spfCycles :- 1	Contains the number of cycles in a zigzag line.
@spfDisp :- 0.5	Contains a value used with various shapes.
@spfFlat :- true	Determines whether the ends of a curved rectangle are flat or a semi-circle.
@spfHeight :- 1.0	Contains the height of a shape.
@spfHorC :- 1.0	Contains the horizontal stretch constant for a quadratic stretch.
@spfHorF :- 0.0	Contains the horizontal stretch factor for a quadratic stretch.
@spfMinorArc :- true	Indicates whether the arc is a major or minor arc in a curved join.
@spfOrigin :- [0.0, 0.0]	Contains a TAC sequence containing the X and Y values by which the calculated path is shifted.
@spfPrecision :- 1	Contains the number of sectors per quadrant contained in the circular arcs of various shapes. A higher value results in a more accurate circular arc. The default value is sufficient in most cases.
@spfRadius :- 1.0	Contains the radius of certain parts of various shapes.
@spfSharpness :- (2.0 / 3.0)	Determines the sharpness of cubic joins or a cubic spline.
@spfSize :- 1.0	Contains the width and height of a square or the length of a standard arrowhead.
@spfVerC :- 1.0	Contains the vertical stretch constant for a quadratic stretch.
@spfVerF :- 0.0	Contains the vertical stretch factor for a quadratic stretch.
@spfWidth :- 1.0	Contains the width of a shape.

3. SPF Commands

The following sections describe the SPF commands.

3.1 Command Summary

The following is a summary of the SPF commands.

SPF Command Summary

Command	Description
@SPF	Creates a new temporary <i>SPF context</i> and returns the <i>current path</i> .
@SPFData	Creates a new <i>SPF context</i> .
spf_CustSPF	Creates and returns a custom SPF procedure that is used like @SPF .
spf_EmbedSPF	Creates and returns a function that calls a specified function within the context of an <i>SPF context</i> .

3.2 Command Definitions

The definitions of the SPF commands are presented below.

@SPF

@SPF *proc* → *path*

proc A procedure stack object.

path A sequence stack object.

Details

Creates a new temporary *SPF context* initialised by a procedure (*proc*), then returns the value of **@spfPath** (*path*).

proc contains calls to the desired SPF functions while the *SPF context* is *active*.

path is the value of **@spfPath** (the *current path*) just before the procedure (*proc*) ends.

The *SPF context* is created with default parameter values and made *active* as long as the procedure (*proc*) is being executed. The *SPF context* is destroyed just before the procedure ends.

Examples

The following illustrations show how the **@SPF** command can be used.

```
vis_GObject("")
{
  [* Create a rounded rectangle item. *]
  vis_CurveItem("")
  {
    [* Specify that the outline only is to be drawn. *]
    @ciFlags := :!GCI_OUTLINED_SHAPE;;

    [* Specify the point sequence of the rounded rectangle. *]
    @ciPoints := @SPF {spf_Rectangle(@spfWidth := 5.0 @spfHeight := 3.0
                                   @spfApexRadius := 1.0)};;
  };
};
```

The code above creates a graphics object that displays a 5.0 by 3.0 rectangle with rounded corners having a radius of 1.0 base unit. The **@SPF** command takes a procedure as its only argument. When the command executes, it first creates a new default-initialised *SPF context* and pushes it onto the TAC dictionary stack. The procedure argument is then activated. The **spf_Rectangle()** function uses the values of the appropriate parameters to create the path of the rectangle into the **@spfPath** parameter. Before finishing, the **@SPF** command returns the value of **@spfPath** (the *current path*) onto the TAC object stack, then removes the *SPF context* from the TAC dictionary, destroying it. The sequence on the TAC stack is then assigned to

@ciPoints. The **spf_Rectangle()** function also uses the **@spfOrigin** parameter. But, by default, that parameter has the value [0.0, 0.0].

It will be noticed that the SPF parameters, in the example above, exist within the parentheses of the **spf_Rectangle()** function. ETAC language syntax allows such to be the case. A more traditional way of calling the function is: **@spfWidth := 5.0; @spfHeight := 3.0; @spfApexRadius := 1.0; spf_Rectangle();**

To illustrate further with another example, the code fragment below creates the same rounded rectangle as above, but rotated 45° ($\pi/4$ radians) in a counter-clockwise direction about the point (1.0, 2.0).

```
...  
[* Specify the point sequence of the rounded rectangle. *]  
@ciPoints := @SPF  
{  
    spf_Rectangle(@spfWidth := 5.0 @spfHeight := 3.0 @spfApexRadius := 1.0);  
    spf_Rotate((:#pi: / 4.0) @spfOrigin := [1.0, 2.0]);  
};  
...
```

The **spf_Rectangle()** function creates the specified rectangle in the **@spfPath** parameter of the *SPF context*, then the **spf_Rotate()** function rotates the value of **@spfPath** as specified, before the **@SPF** command returns the value of **@spfPath** to be assigned to @ciPoints. Note that when the **spf_Rectangle()** function is called, **@spfOrigin** contains its default value.

Other Information

spf_Rectangle() ▀ **spf_Rotate()** ◆

@SPFData

@SPFData *proc* → *spf-data*

proc A procedure stack object.

spf-data An ETAC data object.

Details

Creates a new *SPF context* (*spf-data*) initialised by a procedure (*proc*).

proc contains code to initialise any of the parameters of the new *SPF context*.

spf-data is the new *SPF context* with its parameters initialised by the specified procedure (*proc*). The uninitialised parameters will have default values.

The command takes a procedure argument (*proc*), which typically initialises some of the SPF parameters to the desired values (rather than the default values), and returns the created *SPF context* (*spf-data*) on the TAC stack. The *SPF context* can then be assigned to a variable and used somewhat like the **@SPF** command. For example, the code fragment **DataSPF := @SPFData {...};** creates and assigns an *SPF context* to DataSPF, which is then used somewhat like the **@SPF** command as **... := DataSPF.{...; @spfPath;};**. Note that the use of SPFData can modify its parameter values in subsequent usage.

NOTE: This command must not be called before the first VIS drawing plane has been created.

Example

The following illustration shows how the `@SPFData` command can be used.

```
DataSPF := @SPFData {@spfApexRadius := 1.0;};  
  
...  
  
[* Specify the point sequence of a rounded rectangle. *]  
@ciPoints := DataSPF.{spf_Rectangle(@spfWidth := 5.0 @spfHeight := 3.0); @spfPath;};  
  
...
```

In the example above, the *SPF context* contained in the variable `DataSPF` can be used in multiple places in the ETAC program with the `@spfApexRadius` parameter already set. The `spf_Rectangle()` function uses that set parameter; that is why that parameter is not specified at the function call. Of course, any parameters can be preset as shown and modified at different parts of the program by `DataSPF.{...}` because `DataSPF` contains an ETAC data object. In this example, the parameters `@spfWidth` and `@spfHeight` have been modified in `DataSPF` when creating the rectangle path.

Other Information

`@SPF` ▀ `spf_Rectangle()` ▀ `spf_CustSPF` ♦

spf_CustSPF

`spf_CustSPF` *spf-data* → *proc*

spf-data An ETAC data object.

proc A procedure stack object.

Details

Creates and returns a custom SPF procedure (*proc*) that operates in the same manner as the `@SPF` command does.

spf-data is an *SPF context*, a duplicate of which will be embedded into the returned procedure (*proc*).

proc is a procedure which is used like the `@SPF` command except that it will use the specified *SPF context* (*spf-data*) instead of a new default-initialised *SPF context*.

The returned procedure (*proc*) requires a procedure argument. When the returned procedure is executed, it returns the *current path* (the value of `@spfPath` of the specified *SPF context*).

Example

The following illustration shows how the `spf_CustSPF` command can be used.

```
DataSPF := @SPFData {@spfApexRadius := 1.0;};  
MySPF := spf_CustSPF DataSPF;  
  
...  
  
[* Specify the point sequence of the rounded rectangle. *]  
@ciPoints := MySPF {spf_Rectangle(@spfWidth := 5.0 @spfHeight := 3.0);};  
  
...
```

In the example above, the `spf_CustSPF` command takes an *SPF context* (`DataSPF`) as its only argument, and returns a procedure that can be used like the `@SPF` command. The returned

procedure is assigned to `MySPF`. When called, `MySPF` acts like `@SPF` except that the *SPF context* used is a duplicate of the one contained in `DataSPF`, rather than a new default-initialised one.

Note that `MySPF` pushes a duplicate of the *SPF context* in `DataSPF` onto the dictionary stack when executed. This implies that the original parameter values of the *SPF context* in `DataSPF` remain unaltered on each call to `MySPF`. In this example, `MySPF` is called each time with the `@spfApexRadius` parameter already set via the `@SPFData` command. Any parameter changed within the procedure argument of `MySPF` will be temporary and will not affect the parameter values of `DataSPF`. However, the parameters within `DataSPF` can be changed directly, affecting the duplicated *SPF context* within `MySPF`.

Sometimes it is convenient to create `MySPF` in a single statement if the initialised parameters within `DataSPF` are not going to be altered later: `MySPF := spf_CustSPF @SPFData {...};`.

Other Information

`@SPF` - `@SPFData` - `spf_Rectangle()` ♦

spf_EmbedSPF

spf_EmbedSPF *spf-data fnt* → *new-fnt*

spf-data An ETAC data object.

fnt An ETAC function

new-fnt An ETAC function

Details

Creates and returns an ETAC function (*new-fnt*) that calls a specified function (*fnt*) within the context of a specified *SPF context* (*spf-data*).

spf-data is an *SPF context* which will be embedded into the returned function (*new-fnt*).

fnt is an ETAC function that can call SPF functions without an explicit *SPF context* (the *SPF context* in *spf-data* is used implicitly by *fnt*).

new-fnt is a new ETAC function that calls *fnt* implicitly with the *SPF context* *spf-data*.

When the returned function (*new-fnt*) is called, it returns the *current path* (the value of `@spfPath` of the specified *SPF context*).

Example

The following illustration shows how the `spf_EmbedSPF` command can be used.

```
DataSPF := @SPFData {@spfApexRadius := 1.0;};

RoundedRect := spf_EmbedSPF DataSPF fnt:(pWidth pHeight)
{
  [* This function returns a sequence of points. *]
  spf_Rectangle(@spfWidth := pWidth @spfHeight := pHeight);
};
```

In the example above, the `spf_EmbedSPF` command takes an *SPF context* (`DataSPF`) and a function as its arguments, and embeds the *SPF context* and function into a new returned function (assigned to `RoundedRect`) that is internally defined to call the specified function along with the specified *SPF context*. When the function returned by `spf_EmbedSPF` is called, it returns the *current path* (the value of `@spfPath` of the specified *SPF context*).

The parameters within DataSPF can be changed directly, affecting the *SPF context* embedded within RoundedRectangle. To retain the original parameter values in a different context, DataSPF is duplicated before the call to **spf_EmbedSPF**: `... := spf_EmbedSPF @ DataSPF fnt:(...)` `{...}`;. Alternatively, a new *SPF context* can be created directly to retain the original parameter values: `... := spf_EmbedSPF @SPFData {...} fnt:(...) {...}`;. Note that the embedded *SPF context* is not a duplicate, so the specified function can alter its parameter values. Therefore, if RoundedRectangle is called again, it may have altered parameter values from a previous call to that function.

The RoundedRectangle variable (in the example above) contains a function that takes two arguments (width and height), and is used as shown in the code fragment below.

```
...

[* Assign the point sequence of the rounded rectangle. *]
@ciPoints := RoundedRectangle(5.0 3.0); [* RoundedRectangle() implicitly uses DataSPF. *]

...
```

RoundedRect () calls the **spf_Rectangle ()** function with the *SPF context* contained in DataSPF.

Other Information

@SPFData ▪ spf_Rectangle () ♦

4. SPF Functions

The following sections describe the SPF functions.

4.1 Function Summary

The following is a summary of the SPF functions.

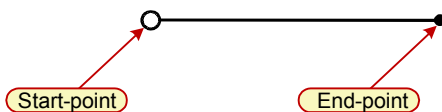
SPF Function Summary

Function	Description
Shape Path Creation Functions	
spf_Brace ()	Creates a left brace.
spf_Circle ()	Creates a circle.
spf_CircleArc ()	Creates an arc of a circle.
spf_CubicJoin ()	Creates a cubic join.
spf_CubicPolyLine ()	Creates a path consisting of joined cubic curves.
spf_CubicSpline ()	Creates a smooth path through a set of points.
spf_CurvedJoin ()	Joins two lines by an arc of a circle.
spf_CurvedRect ()	Creates a curved rectangle.
spf_Diamond ()	Creates a diamond.
spf_Ellipse ()	Creates an ellipse.
spf_EllipseArc ()	Creates an arc of an ellipse.
spf_EquTriangle ()	Creates an equilateral triangle.
spf_IsoTrapezium ()	Creates an isosceles trapezium.
spf_IsoTriangle ()	Creates an isosceles triangle.
spf_Line ()	Creates a line segment.
spf_Oval ()	Creates an oval.

spf_Parabola()	Creates a parabola.
spf_Parallelogram()	Creates a parallelogram.
spf_Polygon()	Creates a regular polygon.
spf_PolyLine()	Creates a path consisting of joined straight lines.
spf_Rectangle()	Creates a rectangle.
spf_RightTriangle()	Creates a right-angled triangle.
spf_Square()	Creates a square.
spf_StdArrowhead()	Creates a path for a standard arrowhead.
spf_Trapezium()	Creates a trapezium.
spf_WavyLine()	Creates a wavy line.
spf_ZigzagLine()	Creates a zigzag line.
Current Path Modification Functions	
spf_Arrowhead()	Creates an arrowhead for the <i>current path</i> .
spf_ClosePath()	Closes the opened <i>current path</i> .
spf_Connect()	Connects a path to the <i>current path</i> .
spf_CvtToLines()	Converts the <i>current path</i> to line segments.
spf_Join()	Joins a path to the <i>current path</i> .
spf_MirrorPath()	Mirrors the <i>current path</i> vertically.
spf_RemExtraPts()	Removes superfluous points from the <i>current path</i> .
spf_RemPoints()	Removes points and their control points from the <i>current path</i> .
Current Path Transformation Functions	
spf_Dilate()	Dilates the <i>current path</i> .
spf_QuadStretch()	Does a quadratic stretch on the <i>current path</i> .
spf_Reflect()	Reflects the <i>current path</i> .
spf_Reverse()	Reverses the <i>current path</i> .
spf_Rotate()	Rotates the <i>current path</i> .
spf_Skew()	Skews the <i>current path</i> .
spf_Translate()	Translates the <i>current path</i> .

4.2 Function Definitions

All numerical values are decimal numbers unless otherwise stated. All paths are *poly-curves* unless otherwise stated. All points are relative to the shape origin (which is the graphics object origin when used with graphics objects) unless otherwise stated. SPF function output is the *current path* unless otherwise stated. In the diagrams associated with the function definitions, the start point of a path is presented as an open circle, and the end point is presented as a filled dot.



4.2.1 Shape Path Creation Functions

In the diagrams of the following entries, the *current path* is shown in black.

spf_Brace()

spf_Brace() [*@spfOrigin* *@spfHeight*]

@spfOrigin A decimal sequence value.

@spfHeight A decimal value.

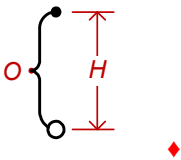
Details

Creates a left brace.

@spfOrigin contains a point ($[x, y]$) indicating the centre of the centre-left of the brace.

@spfHeight contains the height of the brace. This parameter must contain a positive value.

The brace is defined with its centre-left point at *@spfOrigin*. The start point is at the bottom of the brace and the end point is at the top.



spf_Circle()

spf_Circle() [*@spfOrigin* *@spfRadius* *@spfPrecision*]

@spfOrigin A decimal sequence value.

@spfRadius A decimal value.

@spfPrecision An integer value.

Details

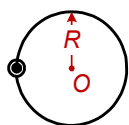
Creates a circle.

@spfOrigin contains a point ($[x, y]$) indicating the centre of the circle.

@spfRadius contains the radius of the circle. This parameter must contain a non-zero value.

@spfPrecision contains the number of sectors per quadrant contained in the circle. This parameter must contain a positive integer.

The circle has its centre at the origin (*@spfOrigin*). The start and end points of the circle are the same and lie horizontally to the left of origin. The circle is defined in a counter-clockwise direction if *@spfRadius* is positive, otherwise it is in a clockwise direction with the start and end points to the right of the origin.



Other Information

spf_CircleArc() ♦

spf_CircleArc()

spf_CircleArc (*pStartPt pEndPt pSegPt*) [*@spfOrigin @spfRadius @spfPrecision*]

pStartPt A decimal sequence stack object.

pEndPt A decimal sequence stack object.

pSegPt A decimal sequence stack object.

@spfOrigin A decimal sequence value.

@spfRadius A decimal value.

@spfPrecision An integer value.

Details

Creates an arc of a circle.

pStartPt is a point ($[x, y]$) defining a line through *@spfOrigin*. The line contains the arc start point. The value of this parameter must not be equal to *@spfOrigin*.

pEndPt is a point defining a line through *@spfOrigin*. The line contains the arc end point. The value of this parameter must not be equal to *@spfOrigin*.

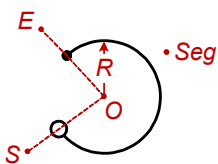
pSegPt is a point ($[x, y]$) in the same extended sector as the major or minor sector of the circle on which the arc is defined. The point determines whether the major or minor arc is requested. The value of this parameter must not be equal to *@spfOrigin* and must not be on a line from *@spfOrigin* to either *pStartPt* or *pEndPt*.

@spfOrigin contains a point ($[x, y]$) indicating the centre of the circle on which the arc is defined.

@spfRadius contains the radius of the circle on which the arc is defined. This parameter must contain a positive value.

@spfPrecision contains the number of sectors per quadrant contained in the arc. This parameter must contain a positive integer.

The arc is defined on a circle with the centre at the origin (*@spfOrigin*). The arc starts at a point determined by *pStartPt* to a point determined by *pEndPt*, and is either the minor arc or major arc depending on the location of *pSegPt*.



Other Information

spf_Circle() ♦

spf_CubicJoin()

spf_CubicJoin(*pStartPt* *pEndPt* *pApexPt*) [*@spfSharpness*]

pStartPt A decimal sequence stack object.

pEndPt A decimal sequence stack object.

pApexPt A decimal sequence stack object.

@spfSharpness A decimal value.

Details

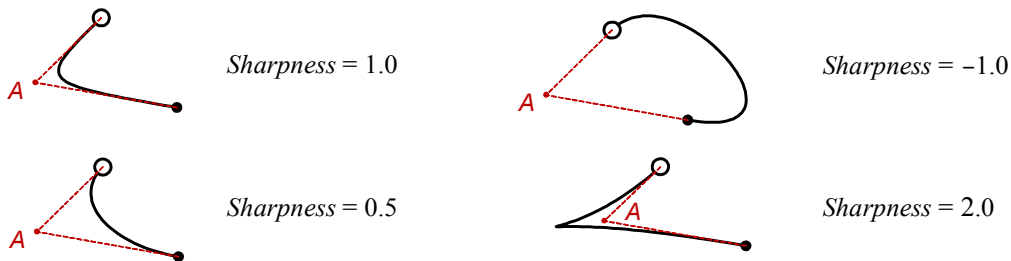
Creates a cubic join.

pStartPt is a point ($[x, y]$) indicating the start point of the path.

pEndPt is a point ($[x, y]$) indicating the end point of the path.

pApexPt is a point ($[x, y]$) defining the intersection of two lines; one from *pStartPt* to *pApexPt* and the other from *pEndPt* to the *pApexPt*.

@spfSharpness contains a value that determines the sharpness of the join. The higher the value the sharper the join. This value can be negative. The default sharpness is 2/3, which results in a parabolic join.



Other Information

spf_CurvedJoin() ♦

spf_CubicPolyLine()

spf_CubicPolyLine(*pPoints*) [*@spfClose* *@spfSharpness*]

pPoints A decimal sequence stack object.

@spfClose An integer value containing a logical boolean value.

@spfSharpness A decimal value.

Details

Creates a path consisting of joined cubic curves.

pPoints is a set of at least three points indicating the vertices of the poly-line. The points have the following format: $[x_1, y_1, x_2, y_2, x_3, y_3[, (x_4, y_4) \dots]]$.

@spfClose contains a boolean value indicating whether the end point of the poly-line is to be connected to the start point by a straight line. A value of `true` indicates that the points are to be connected after the poly-line is created.

@spfSharpness contains a value that determines the sharpness of each cubic join. The higher the value the sharper the join. This value can be negative. The default sharpness is 2/3, which results in parabolic joins.



Other Information

`spf_PolyLine()` ▯ `spf_CubicSpline()` ♦

spf_CubicSpline()

<code>spf_CubicSpline(<i>pPoints</i>) [<i>@spfClose</i> <i>@spfSharpness</i>]</code>	
<i>pPoints</i>	A decimal sequence stack object.
<i>@spfClose</i>	An integer value containing a logical boolean value.
<i>@spfSharpness</i>	A decimal value.

Details

Creates a smooth path through a set of points (interpolative cubic Bezier spline).

pPoints is a set of at least four points through which the path is to pass. The spline will not pass through the first and last points. The points have the following format: [$x_1, y_1, x_2, y_2, x_3, y_3, x_4, y_4$ [, (x_5, y_5) ...]].

@spfClose contains a boolean indicating whether the end point of the resulting spline is to be connected to the start point by a straight line. A value of `true` indicates that the points are to be connected after the spline is created.

@spfSharpness contains a value that determines the sharpness of the spline through the given points. The higher the value the sharper the spline at the given points. This value can be negative. A typical value for *@spfSharpness* is between 0.0 and 1.0. A value of 2.0 for sharpness will create a poly-line. A value greater than 2.0 will create loops at the given points with reduced sharpness. The default sharpness is 2/3.

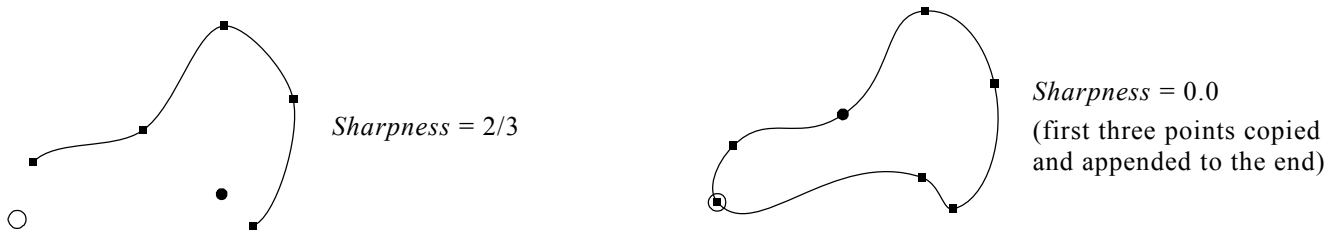
The resulting spline will not pass through the first and last points of *pPoints*, but will pass through the other points. These terminal points, and their adjacent points, define the terminal angles of the spline. The angle between the tangent line at each point (excluding the terminal points) of *pPoints* and each of the adjacent points will be the same.

To get a circle through the four points of a square, copy the first three point to the end and use a *@spfSharpness* value of $(44 - 32 \times \sqrt{2}) / 6$ (≈ -0.209139).

A closed shape can be created by duplicating the first three points, in turn, and appending them to the end of *pPoints*.

To get the spline to pass through the terminal points, repeat those points. A point repeated at least once will produce a sharp corner.

The resulting spline will be at least one-time continuously differentiable ($C^{(1)}$) at each point of *pPoints* (except the end points), but not necessarily more times.



Other Information

spf_CubicPolyLine() ♦

spf_CurvedJoin()

spf_CurvedJoin(<i>pStartPt</i> <i>pEndPt</i> <i>pApexPt</i>) [<i>@spfApexRadius</i> <i>@spfMinorArc</i> <i>@spfPrecision</i>]	
<i>pStartPt</i>	A decimal sequence stack object.
<i>pEndPt</i>	A decimal sequence stack object.
<i>pApexPt</i>	A decimal sequence stack object.
<i>@spfApexRadius</i>	A decimal value.
<i>@spfMinorArc</i>	An integer value containing a logical boolean value.
<i>@spfPrecision</i>	An integer value.

Details

Joins two lines by an arc of a circle.

pStartPt is a point ($[x, y]$) indicating the start point of the path.

pEndPt is a point ($[x, y]$) indicating the end point of the path.

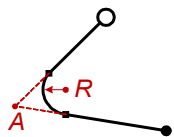
pApexPt is a point ($[x, y]$) defining the intersection of two lines; one from *pStartPt* to *pApexPt* and the other from *pEndPt* to *pApexPt*.

@spfApexRadius contains: (1) The radius of the arc joining the lines from *pStartPt* and *pEndPt* to *pApexPt* if the value is positive. (2) The distance from *pApexPt* to the beginning and to the end of the arc if the value is negative. The absolute value is taken for the distance.

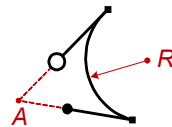
@spfMinorArc contains a boolean value indicating whether the arc is a major or minor arc. If the value is true then a minor arc is requested.

@spfPrecision contains the number of sectors per quadrant contained in the arc. This parameter must contain a positive integer.

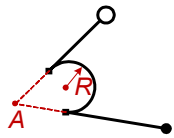
The path starts at *pStartPt* and ends at *pEndPt*. The initial part of the path is a line defined by *pStartPt* and *pApexPt*. The line ends at a point where the arc begins. At that point the tangents of the line and arc are identical. The end part of the path is a line defined by *pEndPt* and *pApexPt*. The line begins where the arc ends. At that point the tangents of the line and arc are identical. The arc is an arc of a circle with either the specified apex radius (if *@spfApexRadius* is positive), or such that the arc begins and ends at the specified distance from *pApexPt* (if *@spfApexRadius* is negative). The arc joins the two lines and is either a major arc or minor arc depending on the value of *@spfMinorArc*. The straight part of the path may be directed away from *pApexPt* depending on the value of *@spfApexRadius*. If *@spfApexRadius* contains 0.0 (zero) then the path consists of a line from *pStartPt* to *pApexPt* to *pEndPt*.



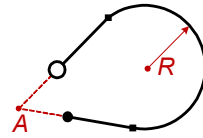
Apex radius (small positive)
Minor arc = true



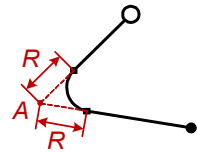
Apex radius (large positive)
Minor arc = true



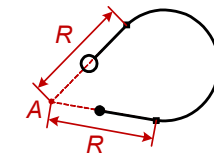
Apex radius (small positive)
Minor arc = false



Apex radius (large positive)
Minor arc = false



Apex radius = -6.8
Minor arc = true



Apex radius = -15.6
Minor arc = false

Other Information

spf_CubicJoin() ♦

spf_CurvedRect()

spf_CurvedRect (*pStartPt pEndPt pSegPt*) [*@spfOrigin @spfRadius @spfWidth @spfFlat @spfPrecision*]

pStartPt A decimal sequence stack object.

pEndPt A decimal sequence stack object.

pSegPt A decimal sequence stack object.

@spfOrigin A decimal sequence value.

@spfRadius A decimal value.

@spfWidth A decimal value.

@spfFlat An integer value containing a logical boolean value.

@spfPrecision An integer value.

Details

Creates a curved rectangle.

pStartPt is a point ($[x, y]$) defining a line through *@spfOrigin*. The line contains the curved rectangle start point. The value of this parameter must not be equal to *@spfOrigin*.

pEndPt is a point ($[x, y]$) defining a line through *@spfOrigin*. The line contains the curved rectangle end point. The value of this parameter must not be equal to *@spfOrigin*.

pSegPt is a point ($[x, y]$) in the same extended sector as the major or minor sector of the circle on which the curved rectangle is defined. The point determines whether the major or minor arc is requested. The value of this parameter must not be equal to *@spfOrigin* and must not be on a line from *@spfOrigin* to either *pStartPt* or *pEndPt*.

@spfOrigin contains a point ($[x, y]$) indicating the centre of the circle on which the curved rectangle is defined.

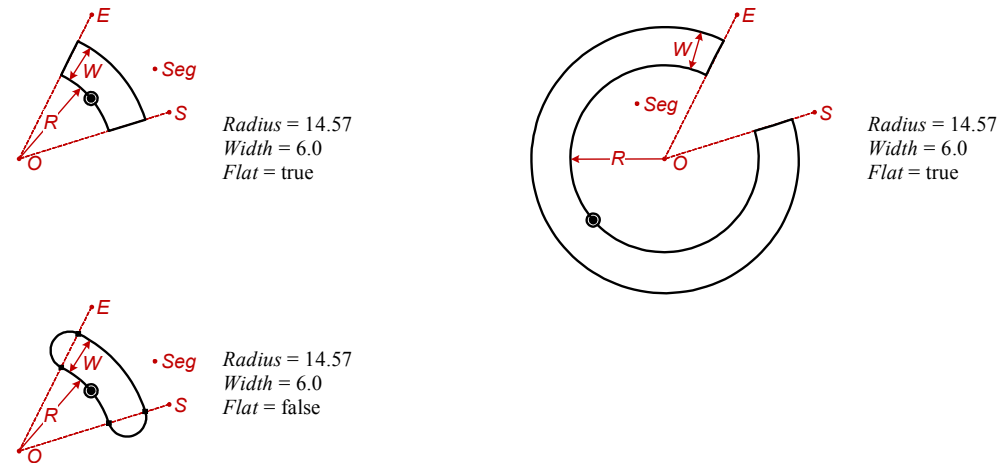
@spfRadius contains the radius of the inner arc of the curved rectangle. This parameter must contain a positive value.

@spfWidth contains the thickness of the curved rectangle. This is the difference between the radii of the outer arc and the inner arc. If *@spfFlat* is `true`, this parameter must contain a non-zero value, otherwise it must contain a positive value.

[@spfFlat](#) contains a boolean value that determines whether the ends of the curved rectangle are flat or a semi-circle. If the value is `true` then the ends of the curved rectangle are straight lines, otherwise they are semi-circles.

[@spfPrecision](#) contains the number of sectors per quadrant contained in each arc of the curved rectangle. This parameter must contain a positive integer.

The curved rectangle is defined on a circle with centre at [@spfOrigin](#). The rectangle starts at a point midway along the inner arc. The initial segment of the path is defined in a direction towards a line defined by [@spfOrigin](#) and [pStartPt](#).



spf_Diamond()	
spf_Diamond() [@spfOrigin @spfWidth @spfHeight @spfApexRadius @spfPrecision]	
@spfOrigin	A decimal sequence value.
@spfWidth	A decimal value.
@spfHeight	A decimal value.
@spfApexRadius	A decimal value.
@spfPrecision	An integer value.

Details

Creates a diamond.

[@spfOrigin](#) contains a point ([x, y]) indicating the centre of the diamond.

[@spfWidth](#) contains the horizontal width of the diamond. This parameter must contain a non-zero value.

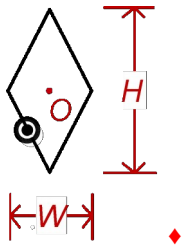
[@spfHeight](#) contains the vertical height of the diamond. This parameter must contain a non-zero value.

[@spfApexRadius](#) contains: (1) The radius of the arc joining the sides of the diamond if the value is positive. (2) The distance from a vertex of the diamond to the beginning and to the end of the arc joining the sides of the diamond if the value is negative. The absolute value is taken for the distance.

[@spfPrecision](#) contains the number of sectors per quadrant contained in the arcs. This parameter must contain positive integer.

The diamond has centre at [@spfOrigin](#). The start and end points of the diamond are the same and lie at the middle of the lower-left side. The path is defined in a counter-clockwise direction if

@spfWidth is positive; the triangle is upside-down if *@spfHeight* is negative. The sides of the diamond are joined by an arc of a circle with a radius of *@spfApexRadius* (if it is positive), or an arc beginning and ending at a distance of negative the value of *@spfApexRadius* (if it is negative) from a vertex. If *@spfApexRadius* contains 0.0 (zero) then the lines join at their vertex.



spf_Ellipse()

spf_Ellipse(<i>pOriPt</i>) [<i>@spfOrigin</i> <i>@spfWidth</i> <i>@spfHeight</i> <i>@spfPrecision</i>]	
<i>pOriPt</i>	A decimal sequence stack object.
<i>@spfOrigin</i>	A decimal sequence value.
<i>@spfWidth</i>	A decimal value.
<i>@spfHeight</i>	A decimal value.
<i>@spfPrecision</i>	An integer value.

Details

Creates an ellipse.

pOriPt is a point ([*x*, *y*]) determining the orientation of the width of the ellipse. This parameter must not be equal to *@spfOrigin*.

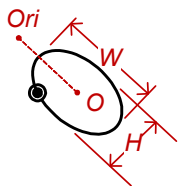
@spfOrigin contains a point ([*x*, *y*]) indicating the centre of the ellipse.

@spfWidth contains the width of the ellipse. The width is along the line defined by *@spfOrigin* and *pOriPt*. This parameter must contain a positive value.

@spfHeight contains the height of the ellipse. The height is perpendicular to the width of the ellipse. This parameter must contain a positive value.

@spfPrecision contains the number of sectors per quadrant contained in the path. This parameter must contain positive integer.

The ellipse has centre at *@spfOrigin*. The start and end points of the ellipse are the same and lie horizontally to the left of *@spfOrigin*. The path is defined in a counter-clockwise direction.



Other Information

spf_EllipseArc() ▀ **spf_Oval()** ♦

spf_EllipseArc()

spf_EllipseArc (*pStartPt pEndPt pOriPt pSegPt*) [*@spfOrigin @spfWidth @spfHeight @spfPrecision*]

pStartPt A decimal sequence stack object.

pEndPt A decimal sequence stack object.

pOriPt A decimal sequence stack object.

pSegPt A decimal sequence stack object.

@spfOrigin A decimal sequence value.

@spfWidth A decimal value.

@spfHeight A decimal value.

@spfPrecision An integer value.

Details

Creates an arc of an ellipse.

pStartPt is a point ($[x, y]$) defining a line through *@spfOrigin*. The line contains the arc start point. This parameter must not be equal to *@spfOrigin*.

pEndPt is a point ($[x, y]$) defining a line through *@spfOrigin*. The line contains the arc end point. This parameter must not be equal to *@spfOrigin*.

pOriPt is a point ($[x, y]$) determining the orientation of the width of the ellipse on which the arc is defined. This parameter must not be equal to *@spfOrigin*.

pSegPt is a point ($[x, y]$) in the same extended sector as the major or minor sector of the ellipse on which the arc is defined. The point determines whether the major or minor arc is requested. The value of this parameter must not be equal to *@spfOrigin* and must not be on a line from *@spfOrigin* to either *pStartPt* or *pEndPt*.

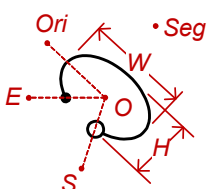
@spfOrigin contains a point ($[x, y]$) indicating the centre of the ellipse on which the arc is defined.

@spfWidth contains the width of the ellipse on which the arc is defined. The width is along the line defined by *@spfOrigin* and *pOriPt*. This parameter must contain a positive value.

@spfHeight contains the height of the ellipse on which the arc is defined. The height is perpendicular to the width of the ellipse. This parameter must contain a positive value.

@spfPrecision contains the number of sectors per quadrant contained in the arc. The parameter must contain a positive integer.

The arc is defined on an ellipse with centre at *@spfOrigin*. The arc starts at a point determined by *pStartPt* to a point determined by *pEndPt*, and is either the minor arc or major arc depending on the location of *pSegPt*.



Other Information

spf_Ellipse() ♦

spf_EquTriangle()

spf_EquTriangle() [*@spfOrigin @spfWidth @spfApexRadius @spfPrecision*]

@spfOrigin A decimal sequence value.

@spfWidth A decimal value.

@spfApexRadius A decimal value.

@spfPrecision An integer value.

Details

Creates an equilateral triangle.

@spfOrigin contains a point ($[x, y]$) indicating the centre of the triangle based on the triangle's width and height.

@spfWidth contains the width of the triangle. Each side has this length. This parameter must contain a non-zero value.

@spfApexRadius contains: (1) The radius of the arc joining the sides of the triangle if the value is positive. (2) The distance from a vertex of the triangle to the beginning and to the end of the arc joining the sides of the triangle if the value is negative. The absolute value is taken for the distance.

@spfPrecision contains the number of sectors per quadrant contained in the arcs. The parameter must contain a positive integer.

The triangle has centre at *@spfOrigin*. The start and end points of the triangle are the same and lie midway along the bottom side of the triangle. The path is defined in a counter-clockwise direction if *@spfWidth* is positive. The sides of the triangle are joined by an arc of a circle with a radius of *@spfApexRadius* (if it is positive), or an arc beginning and ending at a distance of negative the value of *@spfApexRadius* (if it is negative) from a vertex. If *@spfApexRadius* contains 0.0 (zero) then the sides join at their vertex.



Other Information

spf_IsoTriangle() ▪ **spf_RightTriangle()** ♦

spf_IsoTrapezium()

spf_IsoTrapezium() [*@spfOrigin @spfWidth @spfHeight @spfDisp @spfApexRadius @spfPrecision*]

@spfOrigin A decimal sequence value.

@spfWidth A decimal value.

@spfHeight A decimal value.

@spfDisp A decimal value.

@spfApexRadius A decimal value.

@spfPrecision An integer value.

Details

Creates an isosceles trapezium.

@spfOrigin contains a point ($[x, y]$) indicating the centre of the trapezium.

@spfWidth contains the horizontal width of the trapezium. This parameter must not contain a negative value.

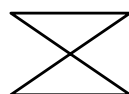
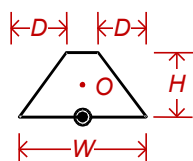
@spfHeight contains the vertical height of the trapezium. This parameter must contain a non-zero value.

@spfDisp contains the positive or negative horizontal distance from the bottom-left vertex point to the top-left vertex point of the trapezium. If the value is negative, then the top-left vertex point is to the left of the bottom-left vertex point. The symmetrical situation occurs for the other two vertex points.

@spfApexRadius contains: (1) The radius of the arc joining the sides of the trapezium if the value is positive. (2) The distance from a vertex of the trapezium to the beginning and to the end of the arc joining the sides of the trapezium if the value is negative. The absolute value is taken for the distance.

@spfPrecision contains the number of sectors per quadrant contained in the arcs. The parameter must contain a positive integer.

The trapezium has centre at *@spfOrigin*. The start and end points of the trapezium are the same and lie midway along the bottom side of the trapezium. The path is defined in a counter-clockwise direction if *@spfHeight* is positive; if *@spfHeight* is negative, the path is upside-down. The sides of the trapezium are joined by an arc of a circle with a radius of *@spfApexRadius* (if it is positive), or an arc beginning and ending at a distance of negative the value of *@spfApexRadius* (if it is negative) from a vertex. If *@spfApexRadius* contains 0.0 (zero) then the lines join at their vertex.



$$@spfDisp = @spfWidth$$

Other Information

spf_Trapezium() ♦

spf_IsoTriangle()

spf_IsoTriangle() [*@spfOrigin @spfWidth @spfHeight @spfApexRadius @spfPrecision*]

@spfOrigin A decimal sequence value.

@spfWidth A decimal value.

@spfHeight A decimal value.

@spfApexRadius A decimal value.

@spfPrecision An integer value.

Details

Creates an isosceles triangle.

@spfOrigin contains a point ($[x, y]$) indicating the centre of the triangle based on the triangle's width and height.

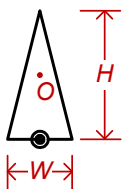
@spfWidth contains the horizontal width of the triangle. This parameter must contain a non-zero value.

@spfHeight contains the vertical height of the triangle. This parameter must contain a non-zero value.

@spfApexRadius contains: (1) The radius of the arc joining the sides of the triangle if the value is positive. (2) The distance from a vertex of the triangle to the beginning and to the end of the arc joining the sides of the triangle if the value is negative. The absolute value is taken for the distance.

@spfPrecision contains the number of sectors per quadrant contained in the arcs. The parameter must contain a positive integer.

The triangle has centre at *@spfOrigin*. The start and end points of the triangle are the same and lie midway along the bottom side of the triangle. The path is defined in a counter-clockwise direction if *@spfWidth* is positive; the triangle is upside-down if *@spfHeight* is negative. The sides of the triangle are joined by an arc of a circle with a radius of *@spfApexRadius* (if it is positive), or an arc beginning and ending at a distance of negative the value of *@spfApexRadius* (if it is negative) from a vertex. If *@spfApexRadius* contains 0.0 (zero) then the sides join at their vertex.



Other Information

spf_EquTriangle() ▪ **spf_RightTriangle()** ♦

spf_Line()

spf_Line (*pStartPt* *pEndPt*)

pStartPt A decimal sequence stack object.

pEndPt A decimal sequence stack object.

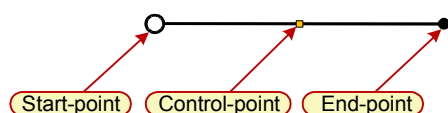
Details

Creates a line segment.

pStartPt is the start point ($[x, y]$) of the line.

pEndPt is the end point ($[x, y]$) of the line.

The straight line starts at *pStartPt* and ends at *pEndPt*. The control points are midway between *pStartPt* and *pEndPt*.



Other Information

spf_PolyLine() ♦

spf_Oval()

spf_Oval () [*@spfOrigin* *@spfWidth* *@spfHeight* *@spfPrecision*]

@spfOrigin A decimal sequence value.

@spfWidth A decimal value.

@spfHeight A decimal value.

@spfPrecision An integer value.

Details

Creates an oval.

@spfOrigin contains a point ($[x, y]$) indicating the centre of the oval.

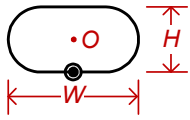
@spfWidth contains the width of the oval. This parameter can contain a negative, zero, or positive value.

@spfHeight contains the height of the oval. This parameter must contain a positive value.

@spfPrecision contains the number of sectors per quadrant contained in the path. The parameter must contain a positive integer.

The oval has centre at *@spfOrigin*. The start and end points of the oval are the same and lie midway along the bottom side of the oval. The path is defined in a counter-clockwise direction. The *@spfWidth* value is the horizontal difference between the middle of the right semicircle path and the middle of the left semicircle path; a value of zero or less will produce a path in the shape of an hourglass.

Note that an oval is not an ellipse. The top and bottom sides of an oval are straight and parallel.



Other Information

spf_Ellipse() ♦

spf_Parabola()

spf_Parabola(*pStartPt* *pEndPt* *pApexPt*)

pStartPt A decimal sequence stack object.

pEndPt A decimal sequence stack object.

pApexPt A decimal sequence stack object.

Details

Creates a parabola.

pStartPt is a point ($[x, y]$) indicating the start of the parabola.

pEndPt is a point ($[x, y]$) indicating the end of the parabola.

pApexPt is a point ($[x, y]$) indicating the apex of the parabola.

The parabola starts at *pStartPt* and ends at *pEndPt*, and intersects *pApexPt*. *pApexPt* is not necessarily the vertex of the parabola, unless *pStartPt*, *pEndPt*, and *pApexPt* form an isosceles triangle with the line segment defined by *pStartPt* and *pEndPt* being the base. The tangent line through *pApexPt* is always parallel to the line defined by *pStartPt* and *pEndPt*. If *pApexPt* is collinear with *pStartPt* and *pEndPt* then the path is a straight line, otherwise it is a segment of a general parabola (ie: a parabola that is not necessarily vertical or horizontal).

To form a parabola corresponding to

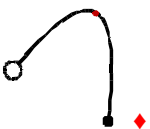
$$y = ax^2 + bx + c,$$

beginning at *pStartPt* and ending at *pEndPt*, the following formula is used to obtain *pApexPt* (*A*).

$$A = ((g - b) / 2a, (g^2 - b^2) / 4a + c),$$

where *g* is the gradient of the line defined by *pStartPt* (*S*) and *pEndPt* (*E*):

$$g = (y_E - y_S) / (x_E - x_S).$$



spf_Parallelogram()

spf_Parallelogram() [*@spfOrigin @spfWidth @spfHeight @spfDisp @spfApexRadius @spfPrecision*]

@spfOrigin A decimal sequence value.

@spfWidth A decimal value.

@spfHeight A decimal value.

@spfDisp A decimal value.

@spfApexRadius A decimal value.

@spfPrecision An integer value.

Details

Creates a parallelogram.

@spfOrigin contains a point ($[x, y]$) indicating the centre of the parallelogram.

@spfWidth contains the base length of the parallelogram (the horizontal distance between a left vertex point and a right vertex point relative to *@spfOrigin*). This parameter must contain a non-zero value.

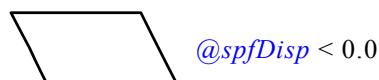
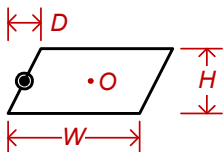
@spfHeight contains the height of the parallelogram perpendicular to the base (the vertical distance between a bottom vertex point and a top vertex point). This parameter must contain a positive value.

@spfDisp contains the positive or negative horizontal distance from the bottom-left vertex point to the top-left vertex point of the parallelogram.

@spfApexRadius contains: (1) The radius of the arc joining the sides of the parallelogram if the value is positive. (2) The distance from a vertex of the parallelogram to the beginning and to the end of the arc joining the sides of the parallelogram if the value is negative. The absolute value is taken for the distance.

@spfPrecision contains the number of sectors per quadrant contained in the arcs. The parameter must contain a positive integer.

The parallelogram has centre at *@spfOrigin*. The start and end points of the path are the same and lie horizontally to the left of *@spfOrigin*. The path is defined in a counter-clockwise direction if *@spfWidth* is positive. The sides of the parallelogram are joined by an arc of a circle with a radius of *@spfApexRadius* (if it is positive), or an arc beginning and ending at a distance of negative the value of *@spfApexRadius* (if it is negative) from a vertex. If *@spfApexRadius* contains 0.0 (zero) then the lines join at their vertex.



Other Information

spf_Rectangle() ♦

spf_Polygon()

spf_Polygon(<i>pNumSides</i>) [<i>@spfOrigin</i> <i>@spfRadius</i> <i>@spfApexRadius</i> <i>@spfPrecision</i>]	
<i>pNumSides</i>	An integer stack object.
<i>@spfOrigin</i>	A decimal sequence value.
<i>@spfRadius</i>	A decimal value.
<i>@spfApexRadius</i>	A decimal value.
<i>@spfPrecision</i>	An integer value.

Details

Creates a regular polygon.

pNumSides is a positive integer specifying the number of sides contained in the polygon. There must be at least three sides.

@spfOrigin contains a point ($[x, y]$) indicating the centre of the polygon.

@spfRadius contains the distance from the centre of the polygon to the furthest point on the polygon. This parameter must contain a non-zero value.

@spfApexRadius contains: (1) The radius of the arc joining the sides of the polygon if the value is positive. (2) The distance from a vertex of the polygon to the beginning and to the end of the arc joining the sides of the polygon if the value is negative. The absolute value is taken for the distance.

@spfPrecision contains the number of sectors per quadrant contained in the arcs. The parameter must contain a positive integer.

The polygon has centre at *@spfOrigin*. The start and end points of the polygon are the same and lie horizontally to the left (if *@spfRadius* is positive) or right (if *@spfRadius* is negative) of *@spfOrigin*. The path is defined in a counter-clockwise direction if *@spfRadius* is positive. The sides of the polygon are joined by an arc of a circle with a radius of *@spfApexRadius* (if it is positive), or an arc beginning and ending at a distance of negative the value of *@spfApexRadius* (if it is negative) from a vertex. If *@spfApexRadius* contains 0.0 (zero) then the sides join at their vertex. The polygon is orientated in such a way that the start and end point are midway along a side.



Sides = 5
Apex radius = 0.0

Other Information

spf_PolyLine() ♦

spf_PolyLine()

spf_PolyLine(*pPoints*) [*@spfClose @spfApexRadius @spfPrecision*]

<i>pPoints</i>	A decimal sequence stack object.
<i>@spfClose</i>	An integer value containing a logical boolean value.
<i>@spfApexRadius</i>	A decimal value.
<i>@spfPrecision</i>	An integer value.

Details

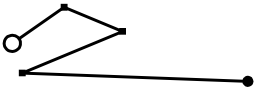
Creates a path consisting of joined straight lines.

pPoints is a sequence of at least two points indicating the vertices of the poly-line. The points have the following format: $[x_1, y_1, x_2, y_2[, (x_3, y_3) \cdots]]$.

@spfClose contains a boolean indicating whether the end point of the poly-line is to be connected to the start point by a straight line. A value of `true` indicates that the points are to be connected after the poly-line is created.

@spfApexRadius contains: (1) The radius of the arc joining the lines of the poly-line if the value is positive. (2) The distance from a vertex of the poly-line to the beginning and to the end of the arc joining lines of the poly-line if the value is negative. The absolute value is taken for the distance.

@spfPrecision contains the number of sectors per quadrant contained in the arcs. The value must contain a positive integer.



Other Information

spf_CubicPolyLine() ▪ **spf_CubicSpline()** ▪ **spf_Polygon()** ▪ **spf_Rectangle()** ♦

spf_Rectangle()

spf_Rectangle() [*@spfOrigin @spfWidth @spfHeight @spfApexRadius @spfPrecision*]

<i>@spfOrigin</i>	A decimal sequence value.
<i>@spfWidth</i>	A decimal value.
<i>@spfHeight</i>	A decimal value.
<i>@spfApexRadius</i>	A decimal value.
<i>@spfPrecision</i>	An integer value.

Details

Creates a rectangle.

@spfOrigin contains a point $([x, y])$ indicating the centre of the rectangle.

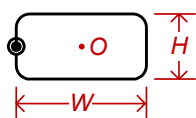
@spfWidth contains the horizontal width of the rectangle. This parameter must contain a non-zero value.

@spfHeight contains the vertical height of the rectangle. This parameter must contain a positive value.

@spfApexRadius contains: (1) The radius of the arc joining the sides of the rectangle if the value is positive. (2) The distance from a vertex of the rectangle to the beginning and to the end of the arc joining the sides of the rectangle if the value is negative. The absolute value is taken for the distance.

@spfPrecision contains the number of sectors per quadrant contained in the arcs. The value must contain a positive integer.

The rectangle has centre at **@spfOrigin**. The start and end points of the rectangle are the same and lie horizontally to the left (if **@spfWidth** is positive) or right (if **@spfWidth** is negative) of **@spfOrigin**. The path is defined in a counter-clockwise direction if **@spfWidth** is positive. The sides of the rectangle are joined by an arc of a circle with a radius of **@spfApexRadius** (if it is positive), or an arc beginning and ending at a distance of negative the value of **@spfApexRadius** (if it is negative) from a vertex. If **@spfApexRadius** contains 0.0 (zero) then the lines join at their vertex.



Apex radius $\neq$ 0.0

Other Information

spf_Square() ♦

spf_RightTriangle()

spf_RightTriangle() [@spfOrigin @spfWidth @spfHeight @spfApexRadius @spfPrecision]	
@spfOrigin	A decimal sequence value.
@spfWidth	A decimal value.
@spfHeight	A decimal value.
@spfApexRadius	A decimal value.
@spfPrecision	An integer value.

Details

Creates a right-angled triangle.

@spfOrigin contains a point ([x, y]) indicating the start and end points of the triangle. The origin is at the right angle.

@spfWidth contains the horizontal distance between **@spfOrigin** and the right angle of the triangle. This parameter must contain a non-zero value.

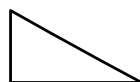
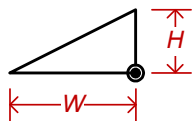
@spfHeight contains the vertical distance between the top apex and the right angle of the triangle. This parameter must contain a non-zero value.

@spfApexRadius contains: (1) The radius of the arc joining the ends of the hypotenuse of the triangle if the value is positive. (2) The distance from a vertex of the triangle to the beginning and to the end of the arc joining the sides of the triangle if the value is negative. The absolute value is taken for the distance.

@spfPrecision contains the number of sectors per quadrant contained in the arcs. The value must contain a positive integer.

The start and end points of the triangle are the same and are the **@spfOrigin** point. The path is defined in a counter-clockwise direction if **@spfWidth** is positive; if **@spfHeight** is negative, the

triangle is upside-down. The sides of the triangle are joined by an arc of a circle with a radius of *@spfApexRadius* (if it is positive), or an arc beginning and ending at a distance of negative the value of *@spfApexRadius* (if it is negative) from a vertex. If *@spfApexRadius* contains 0.0 (zero) then the sides join at their vertex. The right angle is unconditionally joined at its apex.



@spfWidth < 0.0

Other Information

spf_EquTriangle() ▪ **spf_IsoTriangle()** ♦

spf_Square()

spf_Square() [<i>@spfOrigin</i> <i>@spfSize</i> <i>@spfApexRadius</i> <i>@spfPrecision</i>]	
<i>@spfOrigin</i>	A decimal sequence value.
<i>@spfSize</i>	A decimal value.
<i>@spfApexRadius</i>	A decimal value.
<i>@spfPrecision</i>	An integer value.

Details

Creates a square.

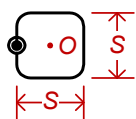
@spfOrigin contains a point ([x, y]) indicating the centre of the square.

@spfSize contains the width and height of the square. This parameter must contain a positive value.

@spfApexRadius contains (1): The radius of the arc joining the sides of the square if the value is positive. (2) The distance from a vertex of the square to the beginning and to the end of the arc joining the sides of the square if the value is negative. The absolute value is taken for the distance.

@spfPrecision contains the number of sectors per quadrant contained in the arcs. The value must contain a positive integer.

The square has centre at *@spfOrigin*. The start and end points of the square are the same and lie horizontally to the left of *@spfOrigin*. The path is defined in a counter-clockwise direction. The sides of the square are joined by an arc of a circle with a radius of *@spfApexRadius* (if it is positive), or an arc beginning and ending at a distance of negative the value of *@spfApexRadius* (if it is negative) from a vertex. If *@spfApexRadius* contains 0.0 (zero) then the lines join at their vertex.



Apex radius ≠ 0.0

Other Information

spf_Rectangle() ♦

spf_StdArrowhead()

spf_StdArrowhead() [*@spfSize*]

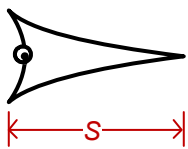
@spfSize A decimal value.

Details

Creates a path for a standard arrowhead.

@spfSize contains the longitudinal length of the arrowhead. This parameter must contain a non-zero value.

The arrowhead points to the right with its vertical baseline intersecting the point (0, 0) and its tip *@spfSize* base units to the right if *@spfSize* is positive, otherwise its tip is to the left from the baseline. The longitudinal length of the arrowhead aligns with the X-axis. The path is defined in a clockwise direction with the start and end points at the base of the arrowhead.



Other Information

spf_Arrowhead() ♦

spf_Trapezium()

spf_Trapezium(*pLength*) [*@spfOrigin @spfWidth @spfHeight @spfDisp @spfApexRadius @spfPrecision*]

pLength A decimal stack object.

@spfOrigin A decimal sequence value.

@spfWidth A decimal value.

@spfHeight A decimal value.

@spfDisp A decimal value.

@spfApexRadius A decimal value.

@spfPrecision An integer value.

Details

Creates a trapezium.

pLength is the length of the top side of the trapezium. The value is the positive or negative horizontal distance from the top-left vertex point to the top-right vertex point of the trapezium.

@spfOrigin contains a point ($[x, y]$) indicating the centre of the trapezium.

@spfWidth contains the horizontal width of the trapezium. This parameter must not contain a negative value.

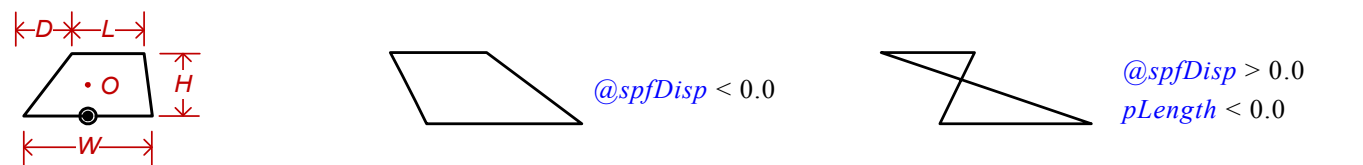
@spfHeight contains the vertical height of the trapezium. This parameter must contain a non-zero value.

@spfDisp contains the positive or negative horizontal distance from the bottom-left vertex point to the top-left vertex point of the trapezium.

@spfApexRadius contains: (1) The radius of the arc joining the sides of the trapezium if the value is positive. (2) The distance from a vertex of the trapezium to the beginning and to the end of the arc joining the sides of the trapezium if the value is negative. The absolute value is taken for the distance.

@spfPrecision contains the number of sectors per quadrant contained in the arcs. The value must contain a positive integer.

The trapezium has centre at *@spfOrigin*. The start and end points of the trapezium are the same and lie midway along the bottom side of the trapezium. The path is defined in a counter-clockwise direction if *@spfHeight* is positive; if *@spfHeight* is negative, the path is upside-down. The sides of the trapezium are joined by an arc of a circle with a radius of *@spfApexRadius* (if it is positive), or an arc beginning and ending at a distance of negative the value of *@spfApexRadius* (if it is negative) from a vertex. If *@spfApexRadius* contains 0.0 (zero) then the lines join at their vertex.



Other Information

spf_IsoTrapezium() ♦

spf_WavyLine()	
spf_WavyLine(<i>pLength</i>) [<i>@spfHeight</i>]	
<i>pLength</i>	A decimal stack object.
<i>@spfHeight</i>	A decimal value.

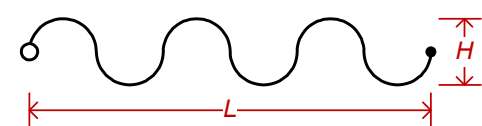
Details

Creates a wavy line.

pLength is the length of the wavy line. This parameter must be a positive value.

@spfHeight contains the maximum height of the wavy line. This parameter must contain a positive value.

The start point of the path is at (0, 0), and is defined along the positive X-axis. The start and end points of the path lie on the X-axis. The number of cycles is such that each wave has a peak-to-peak amplitude as close as possible to, but not exceeding, height. A cycle consists of two semi-circles joined together. The first semi-circle is above the X-axis, and the second one is below the X-axis. A cycle is analogous to a cycle of a sine wave. The shape will be of the specified height (*@spfHeight*) if the length (*pLength*) is divisible by twice the height.



Other Information

spf_ZigzagLine() ♦

spf_ZigzagLine()

spf_ZigzagLine(*pLength*) [*@spfHeight* *@spfCycles* *@spfApexRadius* *@spfPrecision*]

pLength A decimal stack object.

@spfHeight A decimal value.

@spfCycles An integer value.

@spfApexRadius A decimal value.

@spfPrecision An integer value.

Details

Creates a zigzag line.

pLength is the length of the zigzag line. This parameter must be a non-zero value.

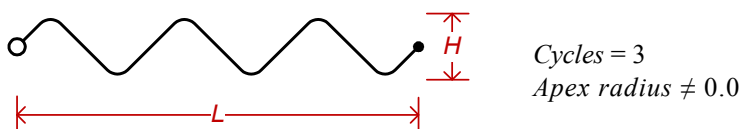
@spfHeight contains the height of the zigzag line. This parameter must contain a non-zero value.

@spfCycles contains the number of cycles in the zigzag line. A cycle starts from the horizontal centre line to the top of the zigzag down to the bottom then to the centre line again. This parameter must contain a positive integer.

@spfApexRadius contains: (1) The radius of the arc joining the lines of the zigzag if the value is positive. (2) The distance from a vertex of the zigzag to the beginning and to the end of the arc joining lines of the zigzag if the value is negative. The absolute value is taken for the distance.

@spfPrecision contains the number of sectors per quadrant contained in the arcs. The value must contain a positive integer.

The start point of the path is at (0, 0), and is defined along the positive X-axis if *pLength* is positive, otherwise along the negative X-axis if *pLength* is negative. The start and end points of the path lie on the X-axis. The straight-line segments within the zigzag are joined by an arc of a circle with a radius of *@spfApexRadius* (if it is positive), or an arc beginning and ending at a distance of negative the value of *@spfApexRadius* (if it is negative) from a vertex. If *@spfApexRadius* contains 0.0 (zero) then the lines join at their vertex. The zigzag is created with *@spfCycles* number of cycles, each cycle being analogous to a cycle of a sine wave. If *@spfHeight* is negative, the path is inverted vertically.



Other Information

spf_WavyLine() ♦

4.2.2 Current Path Modification Functions

In the diagrams of the following entries, the *current path* is shown in black, the specified path is shown in blue, and new paths are shown in green.

spf_Arrowhead()

spf_Arrowhead (<i>pPoints</i>) [<i>@spfAppend</i> <i>@spfClose</i> <i>@spfArrowBase</i> <i>@spfAncPoint</i>] → <i>path</i>	
<i>pPoints</i>	A decimal sequence stack object.
<i>@spfAppend</i>	An integer value containing a logical boolean value.
<i>@spfClose</i>	An integer value containing a logical boolean value.
<i>@spfArrowBase</i>	An integer value.
<i>@spfAncPoint</i>	An integer value.
<i>path</i>	A decimal sequence stack object.

Details

Creates a custom arrowhead aligned to the end of the *current path*.

pPoints is the *poly-curve* path defining the arrowhead. The points have the following format:
 $[x_1, y_1, x_2, y_2, x_3, y_3, x_4, y_4[, (x_5, y_5) \cdots]]$.

@spfAppend contains a boolean value indicating whether the arrowhead is to be connect to the end of the *current path* (true) or not (false).

@spfClose contains a boolean value indicating whether the *current path* is to be closed back along the *current path* to the start point (true) or not (false).

@spfArrowBase contains the desired arrow base number for the arrowhead. The value must be -1 or -2, or a positive integer not exceeding 9.

@spfAncPoint contains the desired anchor point number for the arrowhead. The value must be -1, -2, 0, or a positive integer not exceeding 9. 0 (zero) means to use the same value contained in *@spfArrowBase*.

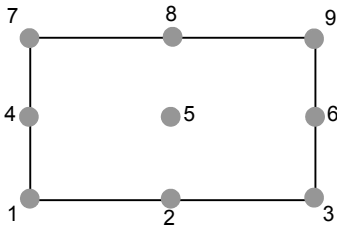
path is the resulting arrowhead path in its proper position in relation to the *current path* if *@spfAppend* contains false, otherwise a copy of the path contained in *pPoints* is returned.

The arrowhead is defined by the path in *pPoints*. It can be any shape but is usually a closed path. The left side of the arrowhead is its “base” and the right side is its “tip” — the arrowhead would therefore be pointing to the right. This function rotates a copy of the arrowhead so that an imaginary horizontal line through it is made parallel with a line through the end point and its control point of the *current path*. The arrowhead copy is then shifted so that the point on the copy indicated by the *@spfArrowBase* parameter (‘arrow base’) will coincide with the end point of the *current path*. The end point and its control point of the *current path* are then shifted together so that the end point will coincide with the point on the copy indicated by the *@spfAncPoint* parameter (‘arrow anchor’). If *@spfAppend* is true then the arrowhead copy is connected to the end of the *current path*, otherwise the arrowhead in *pPoints* is modified so that it is positioned where it would have been connected. If *@spfClose* is true then a copy of the reversed *current path* is appended to the arrowhead if *@spfAppend* is also true.

The values for *@spfArrowBase* and *@spfAncPoint* are as follows:

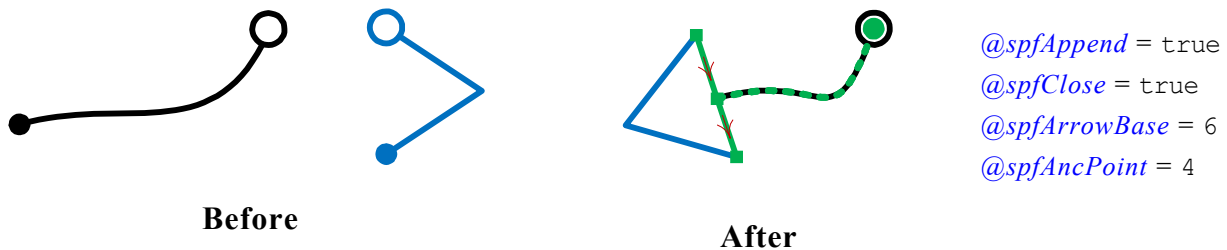
- 1 indicates the start point of the arrowhead path in *pPoints*.
- 2 indicates the end point of the arrowhead path in *pPoints*.
- 0 (not for *@spfArrowBase*) indicates that the *arrow anchor* is the same as the *arrow base*.

The other nine numbers indicate the positions of the arrowhead bounding box as shown in the diagram below (the tip of the arrowhead must be defined to be on the right).



To create a double-ended arrow on the *current path*, the first arrowhead is created with `@spfClose` being `true`, then the second arrowhead is created with `@spfClose` being `false`.

The following diagram is an example of the effects of this function on the *current path* (the black curve on the left) given a specified arrowhead path in *pPoints* (the second blue path from the left). The circle on a path represents the beginning of the path (the circle is not part of the path), and the dot on the other end of a path represents the end of the path (the dot is not part of the path). In the **After** diagram, the arrowhead tip is positioned and the arrowhead lined up with the *current path* end point (`@spfArrowBase = 6`), then the *current path* end point is moved (anchored) to the base of the arrowhead (`@spfAncPoint = 4`). Then the arrowhead path is connected to the end of the *current path* (`@spfAppend = true`), and a reversed copy of the *current path* without the arrowhead (the dotted green path) is appended to the end of the modified *current path* with the arrowhead (`@spfClose = true`). The full path in the third diagram represents the modified *current path*.



Example

The following illustration shows how the `spf_Arrowhead()` function can be used.

```
vis_DPlane("")
{
    vis_GObject("") [* Single-ended path with standard arrowhead. *]
    {
        vis_CurveItem("")
        {
            @ciFlags := :!GCI_OUTLINED_SHAPE;;

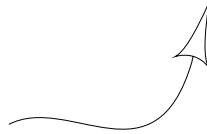
            @ciPoints := @SPF
            {
                @spfPath := [-5.0, 0.0, 0.0, 2.5, 7.0, -5.0, 10.0, 5.0];
                void spf_Arrowhead(@spfArrowBase := 4
                                @SPF{spf_StdArrowhead(@spfSize := 5.0)});
            }
        }
    }
};
```

In the example above, the *current path* (`@spfPath`) is modified to have a standard arrowhead (obtained from the `spf_StdArrowhead()` function) at its end. The value of `@spfArrowBase` (4) means that the middle of the left side of the bounding box of the arrowhead is positioned at the end of the *current path*. The arrowhead parameters not specified take on their default values: `@spfAppend = true` (the end of the *current path* is joined to the beginning of the arrowhead path

by a straight line); `@spfClose = false` (the *current path* is not reversed and joined to the end of the arrowhead path); `@spfAncPoint = 0` (meaning that it is the same as the value of `@spfArrowBase`, ie, 4; the end of the *current path* is not moved).

It will be noticed that the `spf_StdArrowhead()` function needs its own *SPF context*, otherwise the standard arrowhead path will overwrite `@spfPath`.

The single path displayed by the graphics object in the example above is shown below.



Additional Information

5_Creating Custom Arrowheads

Other Information

`spf_StdArrowhead()` ▀ `spf_MirrorPath()` ◆

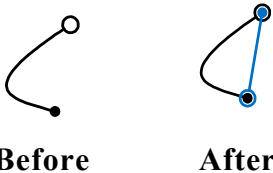
spf_ClosePath()

`spf_ClosePath()`

Details

Closes the opened *current path*.

The *current path* is closed by a straight line. Two control points and the start point are appended to the *current path*. If the *current path* is already closed (the start point and end point are equal) then no action is performed.



Before

After



spf_Connect()

`spf_Connect(pPoints) [@spfAppend]`

pPoints A decimal sequence stack object.

@spfAppend An integer value containing a logical boolean value.

Details

Connects a *poly-curve* path (*pPoints*) to the *current path*.

pPoints is a *poly-curve* path to be connected to the *current path*. The points have the following format: $[x_1, y_1, x_2, y_2, x_3, y_3, x_4, y_4[, (x_5, y_5) \cdots]]$.

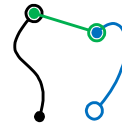
@spfAppend contains a boolean value that determines whether the path in *pPoints* is to be connected to the beginning or end of the *current path*. If the value is `true` then it is connected to the end, otherwise it is connected to the beginning.



Before



After



After

Other Information

`spf_Join()` ♦

spf_CvtToLines()

`spf_CvtToLines()`

Details

Converts the *current path* to line segments.

The inner control points in the *current path* are removed. The end points of the Bezier path segments remain. The illustration below shows two Bezier curves within the *current path*. The Bezier curves become straight lines after the function is called.



Before



After

Note that after calling this function, the *current path* becomes a poly-line not a *poly-curve*.

Other Information

`spf_RemPoints()` ♦

spf_Join()

`spf_Join(pPoints) [@spfAppend]`

pPoints A decimal sequence stack object.

@spfAppend An integer value containing a logical boolean value.

Details

Joins a *poly-curve* path (*pPoints*) to the *current path*.

pPoints is a *poly-curve* path to be joined to the *current path*. The points have the following format: $[x_1, y_1, x_2, y_2, x_3, y_3, x_4, y_4[, (x_5, y_5) \cdots]]$.

@spfAppend contains a boolean value that determines whether the path in *pPoints* is to be joined to the beginning or end of the *current path*. If the value is `true` then it is joined to the end, otherwise it is joined to the beginning.



Before



After



After

Other Information

`spf_Connect()` ♦

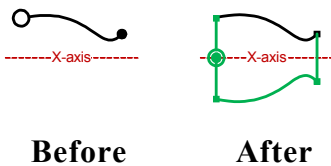
spf_MirrorPath()

`spf_MirrorPath()`

Details

Mirrors the *current path* vertically.

A copy of the *current path* is reflected along the X-axis and connected at both ends to the *current path*, producing a closed path. The start and end points of the closed path lie on the X-axis. The *current path* is defined in the same direction as the original *current path*.



Other Information

`spf_Reflect()` ♦

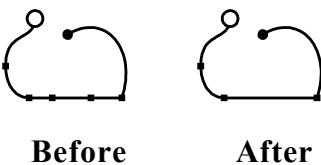
spf_RemExtraPts()

`spf_RemExtraPts()`

Details

Removes superfluous points from the *current path*.

The points that are collinear with the start and end points of each straight-line segment in the *current path* are removed along with their control points.



Other Information

`spf_RemPoints()` ♦

spf_RemPoints()

`spf_RemPoints(pPoints)`

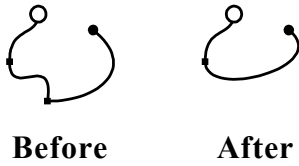
pPoints A decimal sequence stack object.

Details

Removes points and their control points from the *current path*.

pPoints is a set of points (not inner control points) to be removed from the *current path*.

The specified points and their control points in the *current path* are removed. Start and end points in the *current path* have only one control point each; other points have two. The illustration below shows the result of removing the third point (specified in *pPoints*) from the *current path*.



Example

The following illustration shows how the `spf_RemPoints()` function can be used.

```
(1) Path := @SPF{@spfPath := [3.0, 5.2, 4.0, 4.25, 3.1, 4.75, 5.0, 3.3, 2.8, 4.5, 3.85, 3.2, 2.7, 3.1]; spf_RemPoints(5.0, 3.3);};
```

In example (1), the point (5.0, 3.3) is removed from the *current path* (`@spfPath`) resulting in the new *current path* returned in `Path`. The value of `Path` will be: [3.0, 5.2, 4.0, 4.25, 3.85, 3.2, 2.7, 3.1]. The contiguous points (3.1, 4.75), (5.0, 3.3), and (2.8, 4.5) have been removed from the original *current path*.

Other Information

`spf_RemExtraPts()` ♦

4.2.3 Current Path Transformation Functions

The following functions transform the *current path* in various ways.

spf_Dilate()	
<code>spf_Dilate(pHor pVer) [@spfOrigin]</code>	
<i>pHor</i>	A decimal stack object.
<i>pVer</i>	A decimal stack object.
<i>@spfOrigin</i>	A decimal sequence value.

Details

Dilates the *current path*.

pHor is the amount by which to dilate the X-coordinates of the *current path*.

pVer is the amount by which to dilate the Y-coordinates of the *current path*.

@spfOrigin contains the origin point ([x, y]) about which the dilation takes place.

If *@spfOrigin* is (u, v) then each point, (x, y), in the *current path* is modified to:
 $((x - u) \times pHor + u, (y - v) \times pVer + v)$.

A negative value for *pHor* or *pVer* will reflect the shape in a vertical or horizontal line, respectively, through the *@spfOrigin* point.

Other Information

`spf_QuadStretch()` ♦

spf_QuadStretch()

spf_QuadStretch() [*@spfOrigin @spfHorF @spfHorC @spfVerF @spfVerC*]

@spfOrigin A decimal sequence value.

@spfHorF A decimal value.

@spfHorC A decimal value.

@spfVerF A decimal value.

@spfVerC A decimal value.

Details

Does a quadratic stretch on the *current path*.

@spfOrigin contains the origin point ($[x, y]$) about which the stretch takes place.

@spfHorF contains the horizontal stretch factor. The default value is 0.0.

@spfHorC contains the horizontal stretch constant. The default value is 1.0.

@spfVerF contains the vertical stretch factor. The default value is 0.0.

@spfVerC contains the vertical stretch constant. The default value is 1.0.

If *@spfOrigin* is (u, v) then each point, (x, y) , in the *current path* is modified to:

$((x - u) \times (@spfHorF \times |(x - u)| + @spfHorC) + u,$

$(y - v) \times (@spfVerF \times |(y - v)| + @spfVerC) + v).$

For each dimension of the *current path*, the stretch constant applies a dilation about the specified origin; the stretch factor is multiplied by the signed square of the original distance of each point from the origin and the result added to the dilation. The default values of the stretch factor and constant have no effect on the original *current path*.

All kinds of exotic shapes can be created by experimental application of this function with slight modifications of the origin, stretch factor, and stretch constant values from the default values.

Example

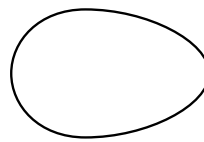
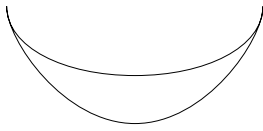
The following illustration shows how the **spf_QuadStretch()** function can be used.

```
vis_GObject("") [* Smile object. *]
{
    vis_CurveItem("")
    {
        @ciFlags := :!GCI_OUTLINED_SHAPE;;

        @ciPoints := @SPF
        {
            spf_Circle(@spfRadius := 3.0);
            spf_QuadStretch(@spfOrigin := [0.0, -1.5] @spfVerF := -0.315);
        };
    };
};
```

In the example above, a quadratic stretch has been applied to a circle to produce the smile shown below on the left. It is the negative vertical stretch factor that makes the top half of the circle flip vertically.

The diagram on the right shows a circle of radius 0.5 units quadratically stretched about the point $(-0.5, 0.0)$ with a horizontal stretch factor of 1.0 and a vertical stretch factor of 0.5.



$@spfOrigin = (-0.5, 0.0)$

$@spfHorF = 1.0$

$@spfVerF = 0.5$

Other Information

spf_Dilate() ♦

spf_Reflect()

spf_Reflect($pStartPt$ $pEndPt$)

$pStartPt$ A decimal sequence stack object.

$pEndPt$ A decimal sequence stack object.

Details

Reflects the *current path*.

$pStartPt$ is the first point $([x, y])$ defining the axis of reflection.

$pEndPt$ is the second point $([x, y])$ defining the axis of reflection.

$pStartPt$ and $pEndPt$ define an axis of reflection. Each point, (x, y) , in the *current path* is modified to: $(2u - x, 2v - y)$ where $(u, v) = pts_Split(pStartPt, pEndPt, (x, y))$. The `pts_split()` function is defined in the documentation `VISPointSetFnts.pdf`.

Other Information

spf_MirrorPath() ♦

spf_Reverse()

spf_Reverse()

Details

Reverses the *current path*.

The sequence of points in the *current path* is reversed. If the *current path* is $[x_1, y_1, x_2, y_2, \dots, x_n, y_n]$, then this function will change it to $[x_n, y_n, x_{n-1}, y_{n-1}, \dots, x_1, y_1]$. ♦

spf_Rotate()

spf_Rotate($pAngle$) [$@spfOrigin$]

$pAngle$ A decimal stack object.

$@spfOrigin$ A decimal sequence value.

Details

Rotates the *current path*.

$pAngle$ is the angle, in radians, that the *current path* is rotated (positive is counter-clockwise).

$@spfOrigin$ contains the origin point $([x, y])$ about which the rotation takes place.

If *@spfOrigin* is (u, v) then each point, (x, y) , in the *current path* is modified to:

$$\begin{aligned} &((x - u) \times \cos(pAngle - (y - v)) \times \sin(pAngle + u), \\ &(x - u) \times \sin(pAngle + (y - v)) \times \cos(pAngle + v)). \blacklozenge \end{aligned}$$

spf_Skew()

spf_Skew(*pHor pVer*) [*@spfOrigin*]

pHor A decimal stack object.

pVer A decimal stack object.

@spfOrigin A decimal sequence value.

Details

Skews the *current path*.

pHor is the amount by which to skew the X-coordinates of the *current path*. A value of zero does not skew the X-coordinates.

pVer is the amount by which to skew the Y-coordinates of the *current path*. A value of zero does not skew the Y-coordinates.

@spfOrigin contains the origin point $([x, y])$ about which the skew takes place.

If *@spfOrigin* is (u, v) then each point, (x, y) , in the *current path* is modified to $((y - v) \times pHor + x, (x - u) \times pVer + y)$. $\blacklozenge$

spf_Translate()

spf_Translate(*pHor pVer*)

pHor A decimal stack object.

pVer A decimal stack object.

Details

Translates the *current path*.

pHor is the amount by which to translate the X-coordinates of the *current path*.

pVer is the amount by which to translate the Y-coordinates of the *current path*.

Each point, (x, y) , in the *current path* is modified to $(x + pHor, y + pVer)$. $\blacklozenge$

5. Creating Custom Arrowheads

Paths with arrowheads typically feature in diagrammatic application programs. In VIS, a programmer-defined arrowhead can optionally be attached to the *current path* using the SPF **spf_Arrowhead()** function which returns a new arrowhead path from the given one. The arrowhead is connected (or positioned) to the end of the *current path* at any one of nine positions relative to the bounding box of the arrowhead, or to the arrowhead's start or end points.

5.1 The Procedure of Positioning an Arrowhead

A program defines an arrowhead (typically by using the SPF functions) as a *poly-curve* of any shape, usually as a closed path. The left side of the arrowhead is its “base” and the right side is its “tip” — the arrowhead would therefore be pointing to the right. The **spf_Arrowhead()** function performs the following procedure after it is called with the arrowhead *poly-curve* path

passed to it. A copy of the arrowhead is positioned correctly at the end of the *current path* as specified by `@spfArrowBase`. The end of the *current path* is then anchored to the arrowhead copy as specified by `@spfAncPoint`. If `@spfAppend` is true then the arrowhead copy is connected to the end of the *current path*, and the original arrowhead is returned. Otherwise, the repositioned arrowhead copy is returned. If both `@spfAppend` and `@spfClose` are true then a copy of the originally shifted *current path* is reversed and then appended to the actual *current path* (containing the arrowhead), resulting in the *current path*'s start and end points being identical; the arrowhead thus appears to be at the other end of the originally shifted path.

The detailed procedure performed by the `spf_Arrowhead()` function is described below. The following table contains definitions relating to that procedure.

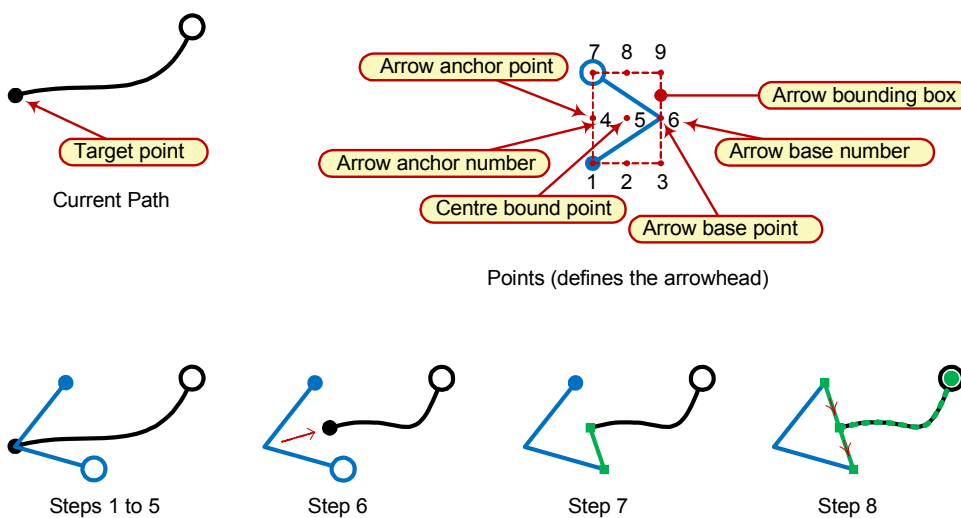
<i>arrow anchor numbers</i>	There are eleven positions for an anchor point for the arrowhead. They are the vertices of the <i>arrow bounding box</i> , the centre point between the vertices, and the <i>centre bound point</i> , the <i>arrow start point</i> , the <i>arrow end point</i> .
<i>arrow anchor point</i>	The point on the arrowhead, indicated by an <i>arrow anchor number</i> , at which the <i>target point</i> will be connected to the arrowhead. The <i>arrow anchor point</i> is relative to the <i>arrow bounding box</i> and the <i>arrow orientation</i> .
<i>arrow base point</i>	The point on the arrowhead, indicated by an <i>arrow anchor number</i> , at which the arrowhead will be positioned relative to the <i>target point</i> before the <i>target point</i> is shifted. The <i>arrow base point</i> is relative to the <i>arrow bounding box</i> and the <i>arrow orientation</i> .
<i>arrow bounding box</i>	The smallest box around the original arrowhead and relative to it containing all the points of the arrowhead.
<i>arrow end point</i>	The current end point of the arrowhead.
<i>arrow lower boundary</i>	The left side of the <i>arrow bounding box</i> .
<i>arrow orientation</i>	The angle (in radians) that the arrowhead has been rotated in a clockwise direction relative to the X-axis.
<i>arrow start point</i>	The current start point of the arrowhead.
<i>arrow upper boundary</i>	The right side of the <i>arrow bounding box</i> .
<i>centre bound point</i>	The point at the centre of the <i>arrow bounding box</i> .
<i>current path</i>	The current set of points which may have been modified from the <i>original path</i> .
<i>original path</i>	The original given set of points defining the <i>current path</i> .
<i>path end point</i>	The last point on the <i>current path</i> .
<i>path start point</i>	The first point on the <i>current path</i> .
<i>reversed end point</i>	The end point of a reversed copy of the <i>current path</i> .
<i>reversed start point</i>	The start point of a reversed copy of the <i>current path</i> .
<i>target point</i>	The point on the <i>current path</i> that is to have an arrowhead put onto it.

The process of positioning an internal copy of the specified arrowhead by the `spf_Arrowhead()` function is as follows:

1. The *arrow bounding box* is calculated.
2. The arrowhead is shifted so that the *centre bound point* is at (0, 0).
3. The arrowhead is rotated so that any horizontal axis through it becomes an axis which is parallel to the *target point* and its control point.
4. The *arrow base point* is set as indicated by the arrow base value (`@spfArrowBase`).

5. The arrowhead is translated (shifted) so that the *arrow base point* coincides with the *target point*.
6. The *target point* and its control point are shifted together to the *arrow anchor point* (indicated by `@spfAncPoint`).
7. If specified (`@spfAppend` is `true`), the arrowhead is connected to the *target point*. Otherwise the arrowhead path is returned by the `spf_Arrowhead()` function with no further processing after the (possibly modified) *current path* becomes the *current path*.
8. If specified (`@spfClose` and `@spfAppend` are both `true`), a reversed copy is made of the (possibly modified) *current path*, and connected to the arrowhead. The `spf_Arrowhead()` function returns the original arrowhead path passed to it after the (possibly modified) *current path* becomes the *current path*.

The diagram below shows the components of a representative arrowhead. The steps described above are also illustrated in the diagram. In this example, the *arrow base point* is 6, and the *arrow anchor point* is 4.



If the arrowhead is not appended to the *current path*, then it can have its own pen and brush attributes; otherwise, the arrowhead and the *current path* together form a single path and can only have the same attributes.

5.2 Examples of Creating an Arrowhead

The following paragraphs present three illustrative examples of paths with arrowheads. Example 1 displays a path with a standard arrowhead attached. Example 2 displays a path with an arrowhead at each end; the path and the two arrowheads form a single closed filled shape. Example 3 shows a path with two separate arrowheads, each with their own path; one of the arrowheads is filled.

5.2.1 Path with a Standard Arrowhead Attached

The code fragment below illustrates how to create a single path containing an arrowhead on the end. The main path is set as the *current path* in the *SPF context*. A standard arrowhead path is then defined (via `spf_StdArrowhead()`) and passed to `spf_Arrowhead()`. The default `@spfAppend` value (`true`) connects (by a straight line) a copy of that arrowhead to the end of the *current path* in the proper position, forming a single path which is then assigned to `@ciPoints`. The `@spfArrowBase` value (4) indicates that the middle of the left border of the arrowhead bounding box is to be connected to the path end. Because the given arrowhead is connect to the *current path*, `spf_Arrowhead()` returns a copy of the arrowhead path passed to it, which is discarded since it is not needed.

```

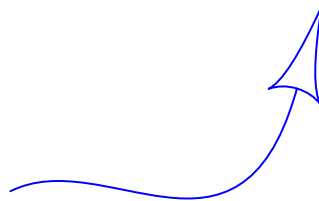
vis_DPlane("")
{
    [* Define the path to have the arrowhead on the end. *]
    dpPath :- [-5.0, 0.0, 0.0, 2.5, 7.0, -5.0, 10.0, 5.0];

    vis_GObject("Example 1") [* Single-ended path with standard arrowhead. *]
    {
        vis_CurveItem("")
        {
            @ciFlags := :!GCI_OUTLINED_SHAPE;;
            vis_Pen("") { @pnColr := vis_RGB(0 0 255); @pnWidth := 0.1;; }

            @ciPoints := @SPF
            {
                @spfPath := dpPath;
                void spf_Arrowhead(@spfArrowBase := 4
                                @SPF{spf_StdArrowhead(@spfSize := 5.0);});
            }
        }
    }
};

```

The code above produces the following shape composed as a single path.



5.2.2 Path with Two Arrowheads Attached

The code fragment below illustrates how to create a single path containing an arrowhead on each end. An arrowhead path is defined and assigned to ArrowheadDef1. Another arrowhead path is defined in the form of a top-half semi-circle ($\cap$) and assigned to ArrowheadDef2. The main path (dpPath) is set as the *current path* in the *SPF context*. The middle of the right border of the first arrowhead bounding box is positioned at the end of the main path (because @spfArrowBase = 6 by default), but the *current path* end is shifted (anchored) to the centre position (because @spfAncPoint = 5) of the arrowhead (ie: the *current path* is modified). The default @spfAppend value (true) connects (by a straight line) a copy of that arrowhead to the end of the *current path* in the proper position, forming a single path. Because @spfClose is true, the end of the arrowhead path is connected to the beginning of a reversed copy of the shifted *current path* (before the arrowhead was appended). As a result, the *current path* has backtracked on itself, therefore its beginning and end points are now identical. Later, the semi-circle arrowhead is connected to the end (which is also the beginning) of the *current path* (because @spfAppend is still true) at the lower-left corner of the original bounding box (because @spfArrowBase and @spfAncPoint both have the value 1). Finally, after all that, the semi-circle end point is connected by a straight line to the end of the *current path* forming a single closed path. That closed path is then assigned to @ciPoints. Note that, because the path is closed, it can be filled with a background colour.

```

vis_DPlane("")
{
    [* Define the path to have the arrowheads on each end. *]
    dpPath :- [-5.0, 0.0, 0.0, 2.5, 7.0, -5.0, 10.0, 5.0];

    [* Filled double-ended closed path with custom arrowheads. *]
    vis_GObject("Example 2")
    {
        vis_CurveItem("")
        {
            start_local;

            [* Create an arrowhead definition. *]
            ArrowheadDef1 :- @SPF{spf_CubicPolyLine([0.0, 2.0, 3.5, 0.0, 0.0, -2.0]
                @spfSharpness := 1.8);};

            [* Create an arrowhead definition in the form of a semi-circle. *]
            ArrowheadDef2 :- @SPF{spf_CircleArc([-1.0, 0.0] [1.0, 0.0] [0.0, 1.0]
                @spfRadius := 1.5);};

            @ciFlags := (:!GCI_OUTLINED_SHAPE: &or :!GCI_FILLED_SHAPE:);

            @ciPoints := @SPF
            {
                @spfPath := dpPath;
                void spf_Arrowhead(ArrowheadDef1 @spfAncPoint := 5 @spfClose := true);
                void spf_Arrowhead(ArrowheadDef2 @spfArrowBase := 1 @spfAncPoint := 1
                    @spfClose := false);

                spf_ClosePath();
            };

            vis_Pen("") {@pnColr := :#RGB_MediumVioletRed; @pnWidth := 0.3;};
            vis_Brush("") {@brColr := :#RGB_SkyBlue;};

            end_local;
        };
    };
};

```

The code above produces the following shape composed as a single filled path.



5.2.3 Path with Two Separate Arrowheads

The code fragment below illustrates how to create two separate arrowheads that are aligned with a path. An arrowhead path is defined in the form of a raindrop with a hole. First, the path of the circle (the hole) is created and reversed (to match up with the rest of the arrow), then assigned to the programmer-defined variable `spfCircle`. Then the raindrop path is created from an isosceles trapezium and rotated so that its start and end points coincide with those of the circle. The two paths are then joined, forming a single closed path as the definition of the raindrop arrowhead assigned to `ArrowheadDef1`. A second arrowhead path is defined and assigned to `ArrowheadDef2`. The main path is set as the *current path* in the *SPF context*. The raindrop arrowhead is then created and set to `Arrowhead1` after being positioned appropriately at the end of the path (`@spfArrowBase := 4`), but not connected to it. The *current path* is now reversed so that the other arrowhead (`Arrowhead2`) can be positioned at the beginning of the original main path but not connected to it. The reversed *current path* may be reversed back if required by the

program. The *current path* is finally assigned to Path. We now have three separate paths in the variables Arrowhead1, Arrowhead2, and Path. Curve items are created from these three paths.

```
vis_DPlane("")
{
    [* Define the path to position the arrowheads at each end. *]
    dpPath :- [-5.0, 0.0, 0.0, 2.5, 7.0, -5.0, 10.0, 5.0];

    vis_GObject("Example 3") [* Path with separate custom arrowheads. *]
    {
        start_local;
        Path :- ?; ArrowheadDef1 :- ?; ArrowheadDef2 :- ?
        Arrowhead1 :- ?; Arrowhead2 :- ?;

        [* Definition for arrowhead 1. *]
        ArrowheadDef1 := @SPF
        {
            spfCircle :- ?;

            spf_Circle(@spfRadius := 1.25); spf_Reverse(); spfCircle := @spfPath;
            spf_IsoTrapezium(@spfWidth := 6.0 @spfHeight := 8.0 @spfDisp := 3.0
                           @spfApexRadius := -3.0);
            spf_Rotate(~(:#pi: / 2.0));
            spf_Join(spfCircle);
        };

        [* Definition for arrowhead 2. *]
        ArrowheadDef2 := @SPF{spf_CubicPolyLine([0.0, 2.0, 3.5, 0.0, 0.0, -2.0]
                                                @spfSharpness := 1.8)};

        [* Create the two arrowheads (Arrowhead1 and Arrowhead2) positioned on the
            ends, but not attached to, the path (Path). *]
        Path := @SPF
        {
            @spfPath := dpPath; @spfAppend := false;
            Arrowhead1 := spf_Arrowhead(ArrowheadDef1 @spfArrowBase := 4);
            spf_Reverse();
            Arrowhead2 := spf_Arrowhead(ArrowheadDef2 @spfArrowBase := 5);
            spf_Reverse(); [* Optional. *]
        };

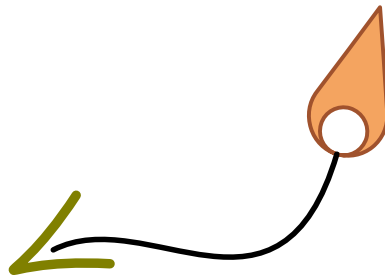
        vis_CurveItem("") [* Arrowhead 1. *]
        {
            @ciFlags := (:!GCI_OUTLINED_SHAPE: &or :!GCI_FILLED_SHAPE:);
            @ciPoints := Arrowhead1;

            vis_Pen("") {@pnColr := :#RGB_Sienna; @pnWidth := 0.2;};
            vis_Brush("") {@brColr := :#RGB_SandyBrown;};
        };

        vis_CurveItem("") [* Arrowhead 2. *]
        {
            @ciFlags := :!GCI_OUTLINED_SHAPE:; @ciPoints := Arrowhead2;
            vis_Pen("") {@pnColr := :#RGB_Olive; @pnWidth := 0.5;};
        };

        vis_CurveItem("") [* Path for arrowheads. *]
        {
            @ciFlags := :!GCI_OUTLINED_SHAPE:; @ciPoints := Path;
            vis_Pen("") {@pnWidth := 0.3;};
        };
        end_local;
    };
};
```

The code above produces the following shape composed of three separate paths.



An arrowhead is a single path, but multiple-path arrowheads can be constructed by creating separate single-path arrowheads and superimposing them with the required modifications.

Bibliography

The Official ETAC Programming Language copyright © Victor Vella (2025).

ETAC: Visual Interaction System copyright © Victor Vella (2025).

MSDN for Visual Studio .NET 2003 copyright 1987-2002 Microsoft Corporation.

Glossary

A

active (refers to an ETAC data object)

This is the first dictionary from the top of the TAC dictionary stack belonging to a referred data object type.

C

current path

This is a TAC sequence of points in the form $[x_1, y_1, x_2, y_2, \dots]$ that is the value of one of the parameters (**@spfPath**) of an *SPF context*, and the object of all SPF functions. The *current path* is a *poly-curve* (with one exception, which is a poly-line) in the form of joined Bezier curve segments. All SPF functions use the *current path*.

P

poly-curve

This is a sequence of X and Y coordinates in the form $[x_1, y_1, x_2, y_2, \dots, x_n, y_n]$, where each $\langle x_i, y_i \rangle$ pair ($i = 1$ to n) represents a point of the *poly-curve*. The *poly-curve* consists of joined Bezier curve segments. Other public documentation needs to be consulted for the definition of a Bezier curve segment. A *poly-curve* must have at least four points (unless it is empty).

S

SPF context

This is an ETAC data object containing pre-defined parameters that are used exclusively by the SPF functions.