VIS Point Set Functions

1 December 2025

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
VISShapePathFnts.pdf	VIS Shape Path 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.

Contents

Document Conventions

The following symbolic conventions are used in this document.

Symbol	Meaning
text	bold green text is a link into the document.
♦	indicates the end of a block of text.

VIS Point Set Functions

This document is for version **1-0-4-ena** of the internal **VIS Point Set 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 VISPointSetFnts.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.

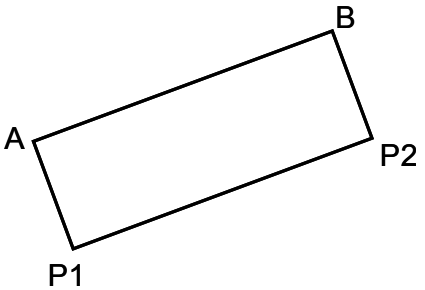
The VIS Point Set Functions library is included in the VIS (**V**isual **I**nteraction **S**ystem) version of the ETAC interpreter, and only requires the inclusion of the VISGlobal.PTAC file to use in ETAC programs.

"PTS" stands for **P**oin**T** **S**et. PTS functions convert given points (each point is a TAC sequence with X and Y values) or values to other points or values. For example, the **`pts_Mult()`** function multiplies a given point by a given value, returning the resultant point. The **`pts_IsPerp()`** function determines whether two position vectors (given as two points) are perpendicular (ie: their dot product is zero).

PTS functions are useful for creating complex diagrams by specifying architectural points on the diagram in relation to each other. Some points are given, and other points are calculated via the PTS functions. The points can then be used to create corresponding lines and curves. For a simple illustration, a rectangle of height `H` can be defined by two given points, `A` and `B`, that determine the length and orientation of the rectangle. The points `A` and `B` are the top-left and top-right points (if `B` is horizontally to the right of `A`). The other two corner points, `P1` and `P2`, can be calculated via PTS functions from `A`, `B`, and `H` as follows:

```
P1 := pts_Vert(A B invtan (H / pts_Dist(A B)) true); P2 := pts_Perp(A B P1).
```

The four points can then be used to define the four sides of the rectangle via the SPF `spf_PolyLine()` function. The diagram below illustrates the rectangle as described.



2. PTS Functions

The following sections describe the PTS functions. Only a small basic set of functions are provided; other functions can be defined by the programmer in terms of the basic set.

2.1 Function Summary

The following is a summary of the PTS functions.

PTS Function Summary

Function	Description
`pts_Add()`	Adds two points.
`pts_Angle()`	Calculates the acute angle between two lines.
`pts_Apex()`	Calculates the vertex point of the right angle opposite the hypotenuse of a right-angled triangle.
`pts_CCWAngle1()`	Calculates the counter-clockwise angle of a line segment relative to the X-axis.
`pts_CCWAngle2()`	Calculates the counter-clockwise angle between two lines.
`pts_Corner()`	Calculates the corner point of two points.
`pts_Dist()`	Calculates the distance between two points.
`pts_Div()`	Divides a point by a value.
`pts_Intsect()`	Calculates the point that intersects two lines.
`pts_IsEqual()`	Determines whether two points are equal.
`pts_IsParallel()`	Determines whether two vectors with the same origin are parallel.
`pts_IsPerp()`	Determines whether two vectors with the same origin are perpendicular.
`pts_Mag()`	Calculates the magnitude of a point.
`pts_Mult()`	Multiplies a point by a value.
`pts_Perp()`	Calculates the point on a line that is perpendicular to another line.
`pts_Proj()`	Calculates the point projected a distance along a line.
`pts_Prop()`	Calculates the point along a line whose length is proportional to the length of the line segment between two points.
`pts_Split()`	Calculates the intersecting point of a line and a perpendicular line through a point.
`pts_Sub()`	Subtracts two points.
`pts_Vert()`	Calculates the vertex point at the end of the hypotenuse of a right-angled triangle with a given side and angle.

2.2 Function Definitions

All *points* are a TAC sequence of the form [*x*, *y*]. The *point* coordinates are decimal numbers.

pts_Add()

`pts_Add(`*pPt1 pPt2*`)` → *output*

pPt1	A decimal sequence stack object.
pPt2	A decimal sequence stack object.
output	A decimal sequence stack object.

Details

Adds two *points* (*pPt1* and *pPt2*).

In the following, *A* is *pPt1*, *B* is *pPt2*.

output is the *point* $(x_A + x_B, y_A + y_B)$.

Other Information

pts_Sub() ♦

pts_Angle()

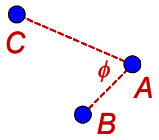
pts_Angle(*pApexPt* *pPt1* *pPt2*) → *output*	
pApexPt	A decimal sequence stack object.
pPt1	A decimal sequence stack object.
pPt2	A decimal sequence stack object.
output	A decimal stack object.

Details

Calculates the acute angle (*output*) between a line defined by two *points* (*pApexPt* and *pPt1*) and another line defined by two *points* (*pApexPt* and *pPt2*).

In the following, A is *pApexPt*, B is *pPt1*, C is *pPt2*, ϕ is *output*.

output (in radians) is $\phi = \cos^{-1}\left(\dfrac{(x_B - x_A)(x_C - x_A) + (y_B - y_A)(y_C - y_A)}{\textbf{pts\_Dist}(A\ B) \times \textbf{pts\_Dist}(A\ C)}\right)$.



Additional Information

pts_Dist()

Other Information

pts_CCWAngle1() ▪ **pts_CCWAngle2()** ♦

pts_Apex()

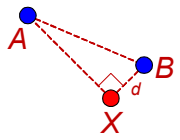
pts_Apex(*pPt1* *pPt2* *pLen* *pRight*) → *output*	
pPt1	A decimal sequence stack object.
pPt2	A decimal sequence stack object.
pLen	A decimal stack object.
pRight	An integer stack object containing a logical boolean value.
output	A decimal sequence stack object.

Details

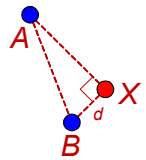
Calculates the vertex *point* (*output*) of the right angle opposite the hypotenuse, defined by two *points* (*pPt1* and *pPt2*), of a right-angled triangle with a side, through the second *point* (*pPt2*), equal to a length (*pLen*). The calculated vertex *point* is on the right (if *pRight* is `true`) or left (if *pRight* is `false`) side of a line segment from the first *point* (*pPt1*) to the second *point* (*pPt2*).

In the following, A is *pPt1*, B is *pPt2*, d is *pLen*, X is *output*.

output is $X = \left(r\left((x_A - x_B)r - (y_A - y_B)\sqrt{1 - r^2}\right) + x_B,\ r\left((y_A - y_B)r + (x_A - x_B)\sqrt{1 - r^2}\right) + y_B\right)$,
where **pts_Dist**(A B) ≥ $|d|$ and $r = \dfrac{d}{\textbf{pts\_Dist}(A\ B)}$ if *pRight* is `true` and d is positive (this is the same as X = **pts_Apex(***A B* –*d* `false`**)**).



output is $X = \left(r\left((x_A - x_B)r + (y_A - y_B)\sqrt{1 - r^2}\right) + x_B,\ r\left((y_A - y_B)r - (x_A - x_B)\sqrt{1 - r^2}\right) + y_B\right)$,
where **pts_Dist**(A B) ≥ $|d|$ and $r = \dfrac{d}{\textbf{pts\_Dist}(A\ B)}$ if *pRight* is `false` and d is positive (this is the same as X = **pts_Apex(***A B* –*d* `true`**)**).



Additional Information
pts_Dist()

Other Information
pts_Perp() ▪ **pts_Corner()** ♦

pts_CCWAngle1()

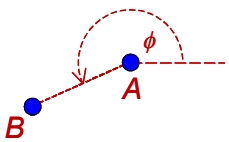
pts_CCWAngle1(*pPt1 pPt2***)** → *output*	
pPt1	A decimal sequence stack object.
pPt2	A decimal sequence stack object.
output	A decimal stack object.

Details
Calculates the counter-clockwise angle (*output*) of a line segment defined by two *points* (*pPt1* and *pPt2*) relative to the X-axis through the first *point*.

In the following, A is *pPt1* and B is *pPt2*, ϕ is *output*.

output (in radians) is $\phi = 2\pi$ – **pts_Angle**(A B $(x_A + 1, y_A)$) if $y_B < y_A$.

output (in radians) is ϕ = **pts_Angle**(A B $(x_A + 1, y_A)$) if $y_B \geq y_A$.



Additional Information
pts_Angle()

Other Information
pts_CCWAngle2() ♦

pts_CCWAngle2()

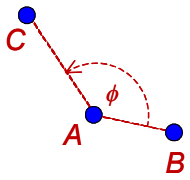
pts_CCWAngle2(*pApexPt pPt1 pPt2***)** → *output*	
pApexPt	A decimal sequence stack object.
pPt1	A decimal sequence stack object.
pPt2	A decimal sequence stack object.
output	A decimal stack object.

Details

Calculates the counter-clockwise angle (*output*) between a line defined by two *points* (*pApexPt* and *pPt1*) and another line defined by two *points* (*pApexPt* and *pPt2*).

In the following, *A* is *pApexPt*, *B* is *pPt1*, *C* is *pPt2*, ϕ is *output*.

output (in radians) is ϕ = **pts_CCWAngle1(***A C***)** – **pts_CCWAngle1(***A B***)** if ϕ is non-negative. If ϕ is negative then 2π is added to ϕ.



Additional Information

pts_CCWAngle1()

Other Information

pts_Angle() ♦

pts_Corner()

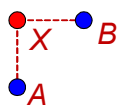
pts_Corner(*pPt1 pPt2***)** → *output*	
pPt1	A decimal sequence stack object.
pPt2	A decimal sequence stack object.
output	A decimal sequence stack object.

Details

Calculates the corner *point* (*output*) of two *points* (*pPt1* and *pPt2*).

In the following, *A* is *pPt1*, *B* is *pPt2*.

output is the *point* $X = (x_A, y_B)$.



Other Information

pts_Perp() ▪ **pts_Apex()** ♦

pts_Dist()

pts_Dist(*pPt1 pPt2***)** → *output*	
pPt1	A decimal sequence stack object.
pPt2	A decimal sequence stack object.
output	A decimal stack object.

Details

Calculates the distance between two *points* (*pPt1* and *pPt2*).

In the following, *A* is *pPt1*, *B* is *pPt2*, *d* is *output*.

output is $d = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$.

Other Information

pts_Mag() ♦

pts_Div()

pts_Div(*pPt pVal***)** → *output*	
pPt	A decimal sequence stack object.
pVal	A decimal stack object.
output	A decimal sequence stack object.

Details

Divides a *point* (*pPt*) by a value (*pVal*).

In the following, *A* is *pPt1*, *V* is *pVal*.

output is the *point* $\left(\frac{x_A}{V}, \frac{y_A}{V}\right)$.

Other Information

pts_Mult() ♦

pts_Intsect()

pts_Intsect(*pPtA1 pPtA2 pPtB1 pPtB2***)** → *output*	
pPtA1	A decimal sequence stack object.
pPtA2	A decimal sequence stack object.
pPtB1	A decimal sequence stack object.
pPtB2	A decimal sequence stack object.
output	A decimal sequence stack object.

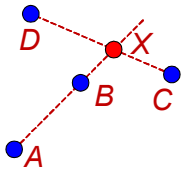
Details

Calculates the *point* (*output*) that intersects the line defined by two *points* (*pPtA1* and *pPtA2*) and the line defined by another two *points* (*pPtB1* and *pPtB2*).

In the following, *A* is *pPtA1*, *B* is *pPtA2*, *C* is *pPtB1*, *D* is *pPtB2*, *X* is *output*.

output is the *point* $X = \left(\frac{y_A - m_1 x_A + m_2 x_C - y_C}{m_2 - m_1}, \frac{m_1(m_2 x_C - y_C) - m_2(m_1 x_A - y_A)}{m_2 - m_1} \right)$,

where $m_1 = \frac{y_B - y_A}{x_B - x_A}$ and $m_2 = \frac{y_D - y_C}{x_D - x_C}$.



Other Information

`pts_Split()` ♦

pts_IsEqual()

pts_IsEqual(*pPt1* *pPt2*) → *output*	
pPt1	A decimal sequence stack object.
pPt2	A decimal sequence stack object.
output	An integer stack object containing a logical boolean value.

Details

Determines whether two *points* (*pPt1* and *pPt2*) are equal.

In the following, *A* is *pPt1*, *B* is *pPt2*.

output is `true` if $x_A = x_B$ and $y_A = y_B$; otherwise it is `false`. ♦

pts_IsParallel()

pts_IsParallel(*pPt1* *pPt2*) → *output*	
pPt1	A decimal sequence stack object.
pPt2	A decimal sequence stack object.
output	An integer stack object containing a logical boolean value.

Details

Determines whether two vectors (*points* *pPt1* and *pPt2*) with the same origin are parallel.

In the following, *A* is *pPt1*, *B* is *pPt2*.

output is `true` if $x_B y_A = x_A y_B$; otherwise it is `false`.

Other Information

`pts_IsPerp()` ♦

pts_IsPerp()

pts_IsPerp(*pPt1 pPt2***)** → *output*	
pPt1	A decimal sequence stack object.
pPt2	A decimal sequence stack object.
output	An integer stack object containing a logical boolean value.

Details

Determines whether two vectors (*points pPt1* and *pPt2*) with the same origin are perpendicular.

In the following, A is *pPt1*, B is *pPt2*.

output is `true` if $x_A x_B + y_A y_B = 0$; otherwise it is `false`.

Other Information

pts_IsParallel() ♦

pts_Mag()

pts_Mag(*pPt***)** → *output*	
pPt	A decimal sequence stack object.
output	A decimal stack object.

Details

Calculates the magnitude of a *point* (*pPt*).

In the following, A is *pPt*, m is *output*.

output is $m = \sqrt{x_A^2 + y_A^2}$.

Other Information

pts_Dist() ♦

pts_Mult()

pts_Mult(*pPt pVal***)** → *output*	
pPt	A decimal sequence stack object.
pVal	A decimal stack object.
output	A decimal sequence stack object.

Details

Multiplies a *point* (*pPt*) by a value (*pVal*).

In the following, A is *pPt1*, V is *pVal*.

output is the *point* (Vx_A, Vy_A).

Other Information

pts_Div() ♦

pts_Perp()

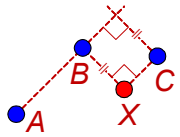
pts_Perp(*pPt1 pPt2 pPt3*) → *output*	
pPt1	A decimal sequence stack object.
pPt2	A decimal sequence stack object.
pPt3	A decimal sequence stack object.
output	A decimal sequence stack object.

Details

Calculates the *point* (*output*) on a line, through the second *point* (*pPt2*), that is perpendicular to the line defined by the first (*pPt1*) and second (*pPt2*) *points*, and of the same length and sense of a perpendicular line segment through the third *point* (*pPt3*).

In the following, A is *pPt1*, B is *pPt2*, C is *pPt3*, X is *output*.

output is the *point* $X = (x_B + x_C - x_1, y_B + y_C - y_1)$, where (x_1, y_1) = **pts_Split**$(A\ B\ C)$.



Additional Information

pts_Split()

Other Information

pts_Corner() · **pts_Apex()** · **pts_Perp()** ♦

pts_Proj()

pts_Proj(*pPt1 pPt2 pDist*) → *output*	
pPt1	A decimal sequence stack object.
pPt2	A decimal sequence stack object.
pDist	A decimal stack object.
output	A decimal sequence stack object.

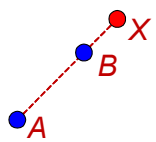
Details

Calculates the *point* (*output*) projected a distance (*pDist*) along a line defined by two *points* (*pPt1* and *pPt2*).

In the following, A is *pPt1*, B is *pPt2*, d is *pDist*, X is *output*.

output is the *point* X = **pts_Prop**$\left(A\ B\ \dfrac{d}{\textbf{pts\_Dist}(A\ B)}\right)$.

d is the distance between *points* A and X. If d is negative, then X is in the opposite direction beyond A.



Additional Information

`pts_Dist()` · `pts_Prop()` ♦

pts_Prop()

`pts_Prop(`*pPt1* *pPt2* *pProp*`)` → *output*	
pPt1	A decimal sequence stack object.
pPt2	A decimal sequence stack object.
pProp	A decimal stack object.
output	A decimal sequence stack object.

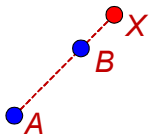
Details

Calculates the *point* (*output*) along a line, defined by two *points* (*pPt1* and *pPt2*), whose length is proportional (*pProp*) to the length of the line segment between the two *points*.

In the following, A is *pPt1*, B is *pPt2*, p is *pProp*, X is *output*.

output is the *point* $X = (x_A + p(x_B - x_A),\ y_A + p(y_B - y_A))$.

p is the distance between A and X divided by the distance between A and B. The *point* midway between A and B is obtained by `pts_Prop(A B 0.5)`. If p is negative, then X is in the opposite direction beyond A.



Other Information

`pts_Proj()` ♦

pts_Split()

`pts_Split(`*pPt1* *pPt2* *pPt3*`)` → *output*	
pPt1	A decimal sequence stack object.
pPt2	A decimal sequence stack object.
pPt3	A decimal sequence stack object.
output	A decimal sequence stack object.

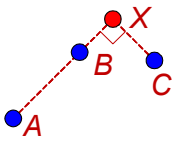
Details

Calculates the intersecting *point* (*output*) of a line defined by two *points* (*pPt1* and *pPt2*) and a perpendicular line through a *point* (*pPt3*).

In the following, A is *pPt1*, B is *pPt2*, C is *pPt3*, X is *output*.

output is the *point* $X = \left(\frac{(x_A - x_B)k}{r} + x_B,\ \frac{(y_A - y_B)k}{r} + y_B\right)$,

where $r = (x_A - x_B)^2 + (y_A - y_B)^2$ and $k = (x_C - x_B)(x_A - x_B) + (y_C - y_B)(y_A - y_B)$.



Other Information

`pts_Perp()` · `pts_Corner()` · `pts_Apex()` ♦

pts_Sub()

`pts_Sub(`*pPt1 pPt2*`)` → *output*	
pPt1	A decimal sequence stack object.
pPt2	A decimal sequence stack object.
output	A decimal sequence stack object.

Details

Subtracts the second *point* (*pPt2*) from the first *point* (*pPt1*).

In the following, A is *pPt1*, B is *pPt2*.

output is the *point* $(x_A - x_B, y_A - y_B)$.

Other Information

`pts_Add()` ♦

pts_Vert()

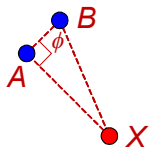
`pts_Vert(`*pPt1 pPt2 pAngle pRight*`)` → *output*	
pPt1	A decimal sequence stack object.
pPt2	A decimal sequence stack object.
pAngle	A decimal stack object.
pRight	An integer stack object containing a logical boolean value.
output	A decimal sequence stack object.

Details

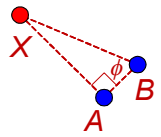
Calculates the vertex *point* (*output*) at the other end of the hypotenuse through a vertex *point* (*pPt2*) of a right-angled triangle with a side through two *points* (*pPt1* and *pPt2*) and an angle (*pAngle*) at the second *point* (*pPt2*). The vertex at the first *point* (*pPt1*) is a right angle. The calculated vertex *point* is on the right (if *pRight* is `true`) or left (if *pRight* is `false`) side of the line segment from the first *point* (*pPt1*) to the second *point* (*pPt2*).

In the following, A is *pPt1*, B is *pPt2*, ϕ is *pAngle* (in radians), X is *output*.

output is $X = (x_A - (y_A - y_B)\tan(\phi),\ y_A + (x_A - x_B)\tan(\phi))$ if *pRight* is `true` and ϕ is positive (this is the same as $X =$ `pts_Vert(`*A B* $-\phi$ `false)`).



output is $X = (x_A + (y_A - y_B)\tan(\phi),\ y_A - (x_A - x_B)\tan(\phi))$ if *pRight* is `false` and ϕ is positive (this is the same as $X =$ **`pts_Vert(`***A B* $-\phi$ `true)`).



Other Information

`pts_Perp()` ♦

3. Skateboard Example

The following is a simple example of using PTS functions to create the side view of a skateboard by specifying architectural points in relation to each other, and drawing the skateboard based on those points.

In the example, points *A*, *B*, and *C* are given. P_1 to P_5 are calculated using the PTS functions. The distance between points *A* and *C* indicates the radius of the wheels. The distance between and positioning of the points *A* and *B* indicate the length and angle of the skateboard. The skateboard can be defined at any angle, even upside-down, by specifying the points *A* and *B* appropriately. All the points are then used to define the image of the skateboard.

The calculations of the points P_1 to P_5 using PTS functions are as follows.

Rad = **`pts_Dist(`***A C***`)`** (this is the radius of each wheel)

P_1 = **`pts_Vert(`***A B* $\tan^{-1}$(Rad / **`pts_Dist(`***A B***`)`**) `true)`

P_2 = **`pts_Proj(`***A* P_1 –(Rad + 0.325)**`)`**

P_3 = **`pts_Perp(`***A B* P_2**`)`**

P_4 = **`pts_Prop(`**P_2 P_3 –0.35**`)`**

P_5 = **`pts_Prop(`**P_3 P_2 –0.35**`)`**

(Note that P_1 is technically unnecessary but is included here for illustration purposes. Without P_1, P_2 would then be: **`pts_Vert(`***A B* $\tan^{-1}$((Rad + 0.325) / **`pts_Dist(`***A B***`)`**) `false)`.) There are other ways of specifying the architectural points depending on the purpose and use of the diagram.

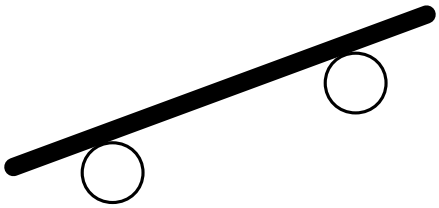
The full program to display the skateboard at an angle is presented below.

```
[*::10ETAC-SOURCE-V1::*]
::include "TACGlobal.PTAC"
::include "VISGlobal.PTAC"

   pop; [* Pop the string argument. *]

start_local;
   vis_Project(?)
   {
      vis_DPlane("")
      {
      A :- [-4.0, -1.5]; B :- [4.0, 1.5]; C :- [-3.0, -1.5]; [* Given points. *]
      Points :- ?;

         void @dpSetSmooth(2); [* Just to give some smoothness to the image. *]

         [* Allow zoom dragging with the right mouse button. *]
         vis_VIEW("MainVIEW") {@vFlags := :!DPV_ZOOM_DRAG:;};

         [* Define the skateboard points, returning the radius and end points. *]
         dpSkateboard :- fnt:(pA pB pC) [* => seq *]
         {
         P1 :- ?; P2 :- ?; P3 :- ?; P4 :- ?;  P5 :- ?; Rad :- ?;

            Rad := pts_Dist(pA pC);
            P1 := pts_Vert(pA pB invtan (Rad / pts_Dist(pA pB)) true);
            P2 := pts_Proj(pA P1 ~(Rad + 0.325));
            P3 := pts_Perp(pA pB P2);
            P4 := pts_Prop(P2 P3 -0.35);
            P5 := pts_Prop(P3 P2 -0.35);

            [Rad, P4, P5]; [*RETURN*]
         };

         [* Create the skateboard points. *]
         Points := dpSkateboard(A B C)
         [* Rad => Points%[1]; P4 => Points%[2]; P5 => Points%[3] *]
```

[continued ...]

```
        [* Create the skateboard graphics object. *]
        vis_GObject("Skateboard")
        {
           vis_LineItem("Board")
           {
              @liFlags := :!GLI_OUTLINED_SHAPE:;
              @liPoints := map_seq `expand_seq [Points%[2], Points%[3]];
              vis_Pen("") {@pnWidth := 0.6;};
           };

           vis_CurveItem("Wheel 1")
           {
              @ciFlags := :!GCI_OUTLINED_SHAPE:;
              @ciPoints := @SPF{spf_Circle(@spfRadius := Points%[1]
                                 @spfOrigin := A);};
              vis_Pen("") {@pnWidth := 0.1;};
           };

           vis_CurveItem("Wheel 2")
           {
              @ciFlags := :!GCI_OUTLINED_SHAPE:;
              @ciPoints := @SPF{spf_Circle(@spfRadius := Points%[1]
                                 @spfOrigin := B);};
              vis_Pen("") {@pnWidth := 0.1;};
           };
        };
     };
  };

  void vis_WaitForEvents("MainVIEW");

end_local;
```

The image produced by the program above is shown below.



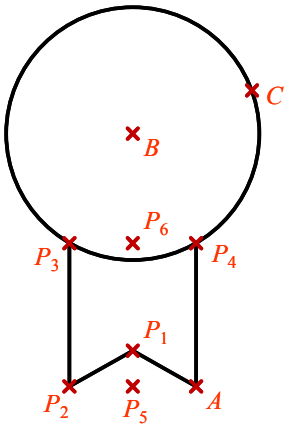
Note that not all graphics images are suitable to be defined in terms of architectural points.

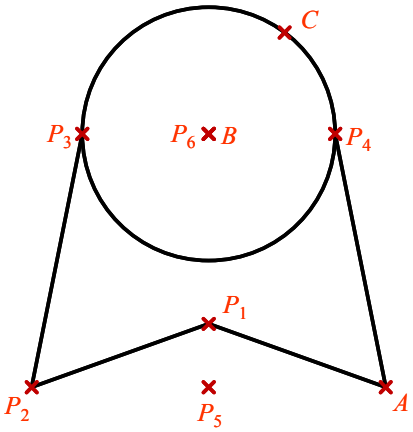
Some graphics programs produce images with handles that can be moved by the user to adjust the features of the image. Such an effect can be achieved via the **`pts_Split()`** and **`pts_Perp()`** functions, where, in both cases, the point *C* would be the handle and the point *X* would be the desired point to be moved along or perpendicular to the line defined by the points *A* and *B* (refer to the diagrams in the definitions of those two functions).

4. Medal Example

The following is an example of using PTS functions to create an image of a medal in terms of architectural points in relation to each other.

In the example, points *A*, *B*, and *C* are given. P_1 to P_6 are calculated using the PTS functions. The distance between points *B* and *C* indicates the radius of the circle. The positioning of the point *A* indicates the width and height of the tag. The left diagram below shows a typical medal; the right diagram shows a medal with the width of the tag greater than the diameter of the circle.





The calculations of the points P_1 to P_6 using PTS functions are as follows.

Rad = **pts_Dist**(*B C*) (this is the radius of the circle)

Width = **pts_Dist**(*A* P_5) (this is half the width of the tag)

Dist = $\sqrt{((\text{Rad}^2) - (\text{Width}^2))}$ [if Rad > Width]
or
0.0 [otherwise]
(this is the distance between *B* and P_6, required to calculate P_4 and P_6)

P_1 = **pts_Prop**(P_5 P_6 0.25)

P_2 = **pts_Prop**(*A* P_5 2.0)

P_3 = **pts_Prop**(P_4 P_6 2.0)

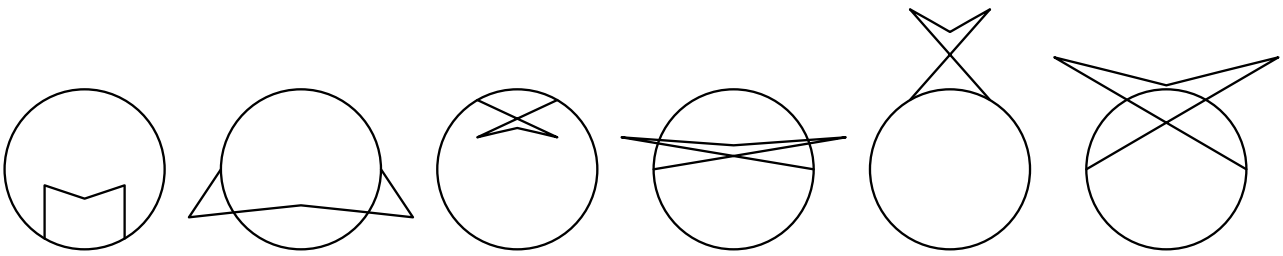
P_4 = **pts_Vert**(P_6 *B* $\tan^{-1}$(Width / Dist) `true`) [if Dist > 0.0]
or
pts_Vert(*B* P_1 $\tan^{-1}$(Rad / **pts_Dist**(*B* P_1)) `false`) [otherwise]

P_5 = **pts_Corner**(*B A*)

P_6 = **pts_Proj**(*B* P_5 Dist)

The order of calculation in a program is as follows: Rad, P_5, Width, Dist, P_6, P_1, P_2, P_4, P_3.

The diagrams below show the image of the medal with various positions of the point *A* above the bottom of the circle and to the right of the centre of the circle. Clearly, some positions produce undesirable images. Proper medal images are produced when the point *A* is below the bottom horizontal line of the circle.



The crossing of the tag can be corrected by specifying *A* to be to the left of and above the centre of the circle. The medal will then appear upside-down.

Bibliography

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

VIS Shape Path Functions copyright © Victor Vella (2025).