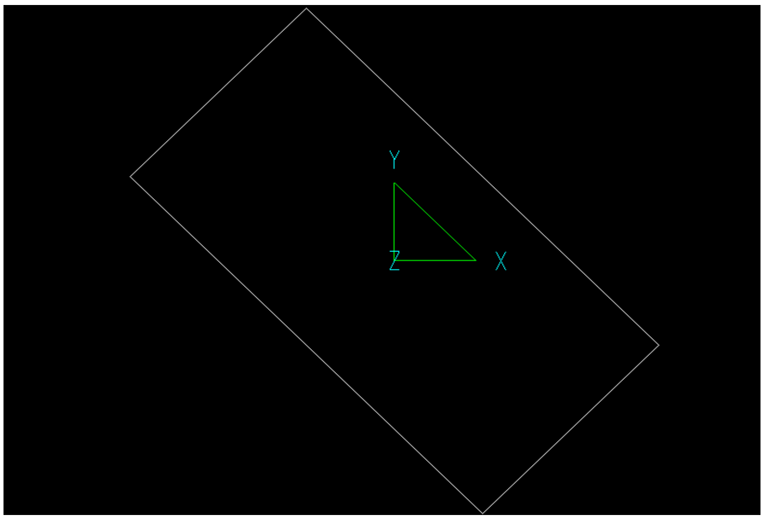
PPM Script Language Reference

Introduction

This reference manual describes the PPM script language. Its attempts to describe the syntax and semantic of the language and provide function overview. PPM script language was created to describe parametric 2D and 3D symbols easily. The language is based on a set of convenient built in functions and a possibility to create custom procedures. Below is a simple example of defining a rectangle as a parametric symbol, with width, height and rotation angle defined through parameters.

```
//Here is a description of a simple rectangle.
H = Parameter("Height",   5, LINEAR, Interval(0, 100));
L =  Parameter("Length", 10, LINEAR, Interval(0, 200));
Angle =  Parameter("Angle", 45, ANGULAR, Interval(0, 360));
Rect1 =  Rectangle(H, L);
Rect =  RotateZ(Rect1, Angle);
Output(Rect);
```

Execution of this script will generate parametric symbol shown in Picture 1.



Picture 1

Let’s examine each line of this example.

First line, which starts with **“//”** (two slashes), contains only a text comment on the example, and it does not affect behavior of the symbol. **“//”** (two slashes) characters mean that all text, starting with this position and up to the end of the line, should be treated as a comment.

Second line defines the characteristics of the **H** parameter. To clarify, let’s spell this line out as an equivalent sequence of lines with comments:

```
H =               // H – identifier (name) of the parameter in the symbol description;
Parameter(        // Parameter – function that defines parameter characteristics;
“Height”,         // This name is shown as a name of the parameter in the Properties dialog;
5,                // 5 - Default parameter value;
LINEAR,           // indicates that H parameter defines a linear value;
Interval(0, 100)  // Defines possible variation interval for the H parameter ;
);                // End of Parameter function and entire description for H.
```

The next two lines in the example are similar to the previous one. They define the characteristics of **L** and **Angle** parameters.

The line **«Rect1 = Rectangle(H, L);»** defines a rectangle with **H** height and **L** length, the center of which lies at the origin of coordinates.

The line **«Rect = RotateZ(Rect1, Angle);»** defines **Rect** rectangle, which is rotated relative to **Rect1** rectangle through **Angle** angle.

The last line in the example - **Output(Rect)**; - shows that **Rect** object is the main (final) in the symbol description.

Script Syntax

Description of a parametric object consists of the entire contents of one or more files, except **comments**, whitespaces, tabs, and other control characters that are ignored.

Text description is a sequence of **statements**, **directives** and **custom functions descriptions.**

Script text may include the following **directives: Input, Output, Include, Units.** Also user can create his/her own functions (**procedures**). Formats of directives and procedures are described in the corresponding sections below.

Comments

A PPM comment can be written in one of the following ways:

- “//” (two slashes) followed by any sequence of characters up to the end of line.
- ”/*” (slash, asterisk) and any subsequence of characters till the “*/” character.

The first way is usually called “single-line comment” and the second is “multiline comment”.

Example:

```
/* This script represents the simplest script, which creates text "Hello, World!" */
PS_TEXT = PropertySet("TFont" = "Times New Roman", "TOblA" = 20);  // Text properties
s = "Hello world!";                                                 // s -  string
txt = Text(PS_TEXT, s, ("Alignment" = "Baseline Left"),  ("THeight" = 100), Point(0, 0), 0);           //txt - 
graphical entity with the text
/* Directive to show the text entity ➔*/ Output(txt);
```

Statements

All **statements** in PPM language have the following format

<Identifier> = <expression>;

Each statement defines identifier with the result of expression.

Identifiers

An identifier **<Identifier>** is a sequence of characters used to denote symbolic name of an object. It is a set of Roman letters and Arabic numerals, which must start with a letter.

For example:

Circle1AR
Rectangle1

Object identifiers may not be the same as names of functions and keywords. These reserved words are used to designate the constants of the scripting language. The list of all reserved names is provided in the reserved word list that appears at the end of this reference manual. Identifiers can contain keywords or function names and they are case sensitive. Therefore, **Rect** and **rect** are different Identifiers.

Expressions

Expression is the main construction in PPM language; it can define an identifier or can be used as part of another expression.

Expression syntax matches the expression syntax in the majority of programming languages.

The simplest expressions are: **Identifiers**, **Real** and **Integer** numbers, **Strings**, **Keywords.**

More complex expressions are created from other expressions (subexpressions) and operations with these subexpressions.

Examples of valid expressions:

0	**-54**	**3.14**	**Alpha**	**"This is a string for testing"**
2 * PI	A + 3 * B	(A + 3) * B	Point(x, 2 * y)	Rectangle(PointX(pt), PointY(pt))

Punctuators

PPM language has a list of punctuators with specific meaning, some of them are used alone and some in pairs.

Punctuator	Meaning
.	decimal symbol
,	to separate function arguments
;	end of the line
()	start and end of list of function arguments
{}	start and end of custom function body
[]	Element of array

Integer numbers

Integer decimal numbers consist of a sign ("+" or "-", optional) and decimal digits (at least one should be present).

Examples of valid integer decimal numbers:

0	**-24**	**+24**	**24**	**1234567**

Integer hexadecimal numbers consist of "0x" and hexadecimal digits (0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f).

Example of valid hexadecimal numbers:

```
0x20     // equal to 32

0x00ff0088
```

Real numbers

As distinct from Integer numbers, real numbers should have at least one of the following parts in the text:

a) Decimal point("."), which separates the integer part from the fractional part, i.e. **3.1415926**.
b) Symbol "e" followed by the power of number 10, i.e. **1e-9**.
c) Measurement unit from the following list:

[in]	**[ft]**	**[mm]**	**[cm]**
[m]	[deg]	[rad]	

Examples of valid real numbers:

0.5	**-0.5**	**0.0**	**1.2345e6[m]**

Strings

String is a set of characters bound by " character. For example "Test string.".

Keywords

Keywords are predefined reserved identifiers that have special meanings. They cannot be used as identifiers in your program.

Here is the list of Keywords:

Keywords	Description
LINEAR	Means that type of parameter is linear in selected linear units of measure.
ANGULAR	Means that type of parameter is angular in selected angular units of measure.
TEXT	Means that type of parameter is text.
FONT	Means that parameter is font name.
COLOR	Means that type of parameter is RGB color.
MATERIAL	Means that parameter is material name.
CHECKBOX	Means that parameter is logical value, either ON or OFF.
LOCK_X	Locks X coordinate of parameter point.
LOCK_Y	Locks X coordinate of parameter point.
DM_CIRCLE	Sets the draw mode of parameter point in circle.
DM_SQUARE	Sets the draw mode of parameter point in square.
DM_XARROW	Sets the draw mode of parameter point in x-arrow.
DM_YARROW	Sets the draw mode of parameter point in y-arrow.
LEFT	Used to set text justification.
CENTER	Used to set text justification.
RIGHT	Used to set text justification.
TOP	Used to set text justification.
MIDDLE	Used to set text justification.
BASELINE	Used to set text justification.
BOTTOM	Used to set text justification.
BOLD	Used to define text style.
ITALIC	Used to define text style.
UNDERLINE	Used to define text style.
BOX	Used to define text style. Not supported.
ALLCAPS	Used to define text style. Not supported.

STRCKETTHROUGH	Used to define text style. Not supported.
INTEGER	Reserved.
REAL	Reserved.
TIME	Reserved.
BRUSH	Reserved.
PCS	Reserved.

Operators

Complex expressions use operators to define arithmetic, logical and other operations.

Operator	Operation	Example	Valid types of operands	Type of Result
+	Add	a + 5	Real, Integer, Strings, Points	As operands
-	Subtract	a - 5	Real, Integer, Points	As operands
*	Multiply	2*PI	Real, Integer, Points	As operands
/	Division	PI/2	Real, Integer	As operands
**	Power	R**4	Real**Integer	As operands
==	Equal	bRect = (w==H)	Real, Integer	Boolean
>	Greater	a > b	Real, Integer	Boolean
>=	Greater or Equal	a >= b	Real, Integer	Boolean
<	Less	a < b	Real, Integer	Boolean
<=	Less or Equal,	a <= b	Real, Integer	Boolean
!=	Not Equal	a != b	Real, Integer	Boolean
&	And	(a < b) & (c < d)	Boolean	Boolean
||	Or	(a < b) || (c < d)	Boolean	Boolean
[]	Element of array	g[i]	Array[Integer]	

Functions

PPM language has a list of built in functions that can be called in the expressions. Also, it is possible to create custom functions in order to simplify the scripts. All function calls have the following format:

<Function name>(<List of arguments with commas as delimiters>)

Examples of valid function calls:

Point(x, 0.05) **sin(PI/6)** **PropertSet("PenColor" = RGB(255, 0, 0), "PenWidth" = 0.3)**

The list of all built-in PPM functions is shown in Function Reference topic. Here is a table of math functions that can be used in PPM scripts:

Name	Result	Examples*
abs(R)	Absolute value	dx = abs(x1 - x2)
arccos(x)	Angle (a), so cos(a) is the specified x	a = arccos(d)
arcsin(x)	Angle (a), so sin(a) is the specified x	a = arcsin(d)
atan(x)	Arctangent of x	a = atan(dy/dx) // dx != 0
atan2(y, x)	Arctangent of (y/x)	Arctangent of (y/x), a = atan2(dy, dx)
cos(a)	Cosine of the specified angle	cs = cos(a)
exp(x)	Exponential value of argument	e = exp(h)
int(x)	Integer value after removing fractional part of argument	k = int(R) //convert to integer (if R = 7.8 then k =7)
ln(x)	Natural logarithm of x	lnR = ln(R)
max(…)	The function is used for choosing maximum values within a set of values	R=max(2, 5, 1, 7, 9)//R=9
min(…)	The function is used for choosing the minimum values within a set of values	r=min(2, 5, 1, 7, 9)//r=1
Mod(a, b)	remainder of the integer division	Mod(5, 4)// is 1
mod(R, r)	The remainder after dividing R by r.	mod(a,b)
round(R, r)	Rounds R where (r) defines the max number (r/2) rounding to zero.	Sr= round(S, 0.1)// 1.23 => 1.2; 1.26 => 1.3
sign(x)	1.0 if x>0; 0 if x== 0.0; -1.0 if x<0.	x =abs(x1) * sign(x2)// take value from x1 and sign from x2
sin(a)	Sine of the specified angle	x = x * sin(alpha)
sqrt(x)	Square root of x	length = sqrt(x * x + y * y)
tan(a)	Tangent of the specified angle	y = x * tan(alpha)

Directives

There are three types of Directives in PPM script: **Include**, **Output**, **Units**.

Include

Include directive is used to include external text file into the current script file.

Format:

```
Include(<relative or absolute path to the file to be included>);
```

Examples:

```
Include("MyFunctions.ppm");
Include("d:\Styles\Styles1.ppm");
```

Output

Output directive defines variables and expressions to be calculated and presented outside of the script.

Format:

```
Output(<List of output variables and expressions with commas as delimiters>);
```

Examples:

```
Output(Roof, Walls,Door1);
Output(graphic1, RotateZ(graphic1, 45[deg]));
```

Input

Input directive defines the list of scripts arguments.

Format:
Input(<List of input arguments separated with commas>);

Examples:
Input(Text, Width, Heigth);

Units
Units directive defines linear and angular measurement units for numbers in the script. By default, units for linear variables are inches, and degrees – for angular variables. It means that, for instance, sin(5) is considered as sin of 5 degrees, and Point(2,0) is a point that lies 2 inches right from origin Point(0,0).

Units directive changes these defaults units.

Format:
Units(1[<linear unit>], 1[<angular unit>]);

List of possible units:

Linear	**[in]**	**[ft]**	**[mm]**	**[cm]**	**[m]**
Angular	[deg]	[rad]			

Example:
Units(1[mm], 1[rad]);

Note: All numbers in the script defined with units will be transformed to default units before calculation.

Thus the next fragment

```
Units(1[mm], 1[rad]);
Alpha = 5[deg]; Len = 1[in];
```

is equivalent to:

```
Units(1[mm], 1[rad]);
Alpha = 0.0872664625; Len = 25.4;
```

Script Semantic
A script should contain complete description of the parametric object. The collection of script operators determines what actions need to be performed to create the resultant object(s). Correct understanding of a script requires having a clear understanding of how its operators are interpreted.

A correct script describing a parametric part should conform to the following rules:

1. Each statement should end with ";" character.

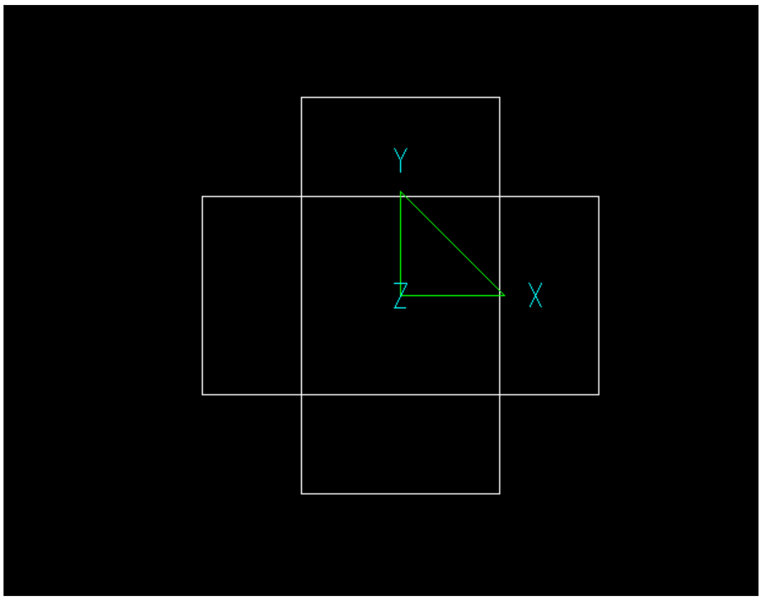
   Rectangle(10,5);

2. Identifiers that are used in expressions should be defined with any value or expression.

   <Identifier> = <Expression>;

3. The list of resultant objects is defined in the Output(..) directive. The Output(..) directive contains a list of objects that are to be displayed in the resulting part. This directive must be present in the script. At least one object must be listed in the Output directive. Expressions and identifiers can be its operands.

   ```
   rect = Rectangle(10, 5);
   Output(rect, Rectangle(5, 10));
   ```



Picture 2

4. A script may have more than one Output(..) directive, but any <Identifier> should be used in only one Output(..) directive.

```
rect = Rectangle(10, 5);
Output(rect);
Output(Rectangle(5, 10));
```

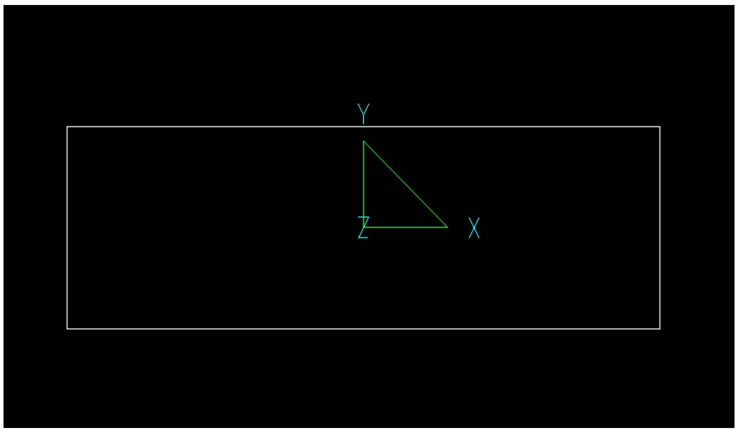
Script above will give the same result as the script in paragraph 3.

5. Identifier can be used in an expression before being defined. The following example is correct:

```
Output(rect);
rect = Rectangle(10, 5);
```

6. Each identifier can be defined many times, but only the last definition will be used.

```
W = 5;
rect = Rectangle(W,5);
W =10;
Output(rect);
W = 15;
```



Picture 3

The identifier **W** will be defined with 15.

7. Circular calculation and interdependent referencing are not allowed. The script must not contain interdependencies where “Item One” is defined by “Item Two”, and “Item Two” is defined by “Item One”. The following script is not correct and application that translates the script will show an error or skips these lines.

   ```
   A = B + 15;
   B = C + 5;
   C = A * 2;
   ```

   In this script, A is defined by C through B, and C is defined by A. In other words, an identifier is not allowed to depend on itself. For example, the following script is also incorrect:

   ```
   A  = A * 2;
   ```

8. The sequence of script operators is not important (except in certain special cases that will be described later); because operators are sorted before the script is run.
9. The default units of script are inches for linear values and degrees for angular. Units directive allows defining other units for the entire script.

   ```
   Units(1[m]);
   Output(Rectangle(5,10));
   ```

   The resultant rectangle will have the width of 5 meters and the height of 10 meters. The following example shows how to specify measure units for a value directly:

   ```
   Units(1[m]);
   Output(Rectangle(5, 10[mm]));
   ```

   The resultant rectangle will have the width of 5 meters and the height of 10 millimeters.

Function Reference

Probably the most significant advantage of this method of creating parametric parts is the compact size and clarity of the text description of parametric parts in script form. The set of basic functions used in such a description determines the level of clarity and simplicity of scripts for a particular class of parametric parts.

Predefined Functions

PPM language contains the list of predefined function. Those functions can be used to describe a parametric object.

Functions for External Parameters

Parameter

Defines parameter characteristics.

Call format:

```
<id> = Parameter(<name>, <value>[, <type>, <conditions>]);
```

Note: The ‘<>’ markers are used to designate elements in the expression, and the ‘[ ]’ markers are used to indicate arguments that are optional.

Arguments:

name – The parameter name;
value - The parameter default value;
type - The parameter type. The following types are available:
LINEAR - numerical value in the selected linear units of measure;

ANGULAR - numerical value in the selected angular units of measure;
TEXT - text string;
FONT - font name;
COLOR - RGB color value;
MATERIAL - material name;
CHECKBOX - logical value, either On or Off

conditions – Optional components defining parameter restrictions. The following restrictions are available and can be specified in any order:

Set(<value>,...) — a list of permissible values of the parameter
Interval(<minvalue>, <maxvalue>) — sets the minimum and maximum values of the parameter;
LessThan(<value>) — indicates that parameter value should be less than the specified value
LessOrEqual(<value>) — indicates that parameter value should not be greater than the specified value
GreaterThan(<value>) — indicates that parameter value should be greater than the specified value
GreaterOrEqual(<value>) — indicates that parameter value should not be smaller than the specified value
Set(FolderList) — particular case of Set operator, when a list of permissible values are defined by function FolderList.

Examples of use:

```
clr = Parameter("Color", 0xff, COLOR);
fnt = Parameter("Font", "Arial", FONT);
H = Parameter("Height", 36, LINEAR, Set(30, 36, 42));
W = Parameter("Width", 18, LINEAR, LessThan(15));
D = Parameter("Depth", 14, LINEAR, Interval(10,15));
Angle = Parameter("Angle", 45, ANGULAR, Interval(0, 360));
txt = Text("Simple text example", TextFont(0,1,0, fnt), TextStyle(CENTER, MIDDLE, ITALIC, ALLCAPS));
txtClr = SetProperties(txt, "PenColor" = clr);
id = Thickness(Rectangle(H, W), D);
idRotated =  RotateZ(id, Angle);
Output(txtClr, idRotated);
```

ParameterPoint

Defines a parametric point.

Call format:

```
<id> = ParameterPoint(<point>);
<id> = ParameterPoint(<index>, <x>, <y>, <z>);
<id> = ParameterPoint(<name>, <x>, <y> [, <BasePoint>, <Matrix>, <Style>,< Fixed>, <DrawMode>, <Color>]);
<id> = ParameterPoint(<name>, <x>, <y> [,<BasePoint1>, <BasePoint2>,<Style>, <Fixed>, <DrawMode>, Color>]);
```

Arguments:

point – The default value of parameter point.
index – The index of parameter point.
x - X coordinate of the point.
y - Y coordinate of the point.
z - Z coordinate of the point.
name – The parameter point name in user interface.
Fixed – Sets locked coordinate of parameter point coordinate: LOCK_X or LOCK_Y.

BasePoint – Sets the base point in the form: ID or Point(bpX, bpY). The current value will be: BasePoint * Matrix + Point(x, y).
BasePoint1, BasePoint2 - Base points in the form: ID or Point(bpX, bpY). Two points that define a Matrix.
Matrix - Matrix that will be applied to the base point.
Style - Defines the parametric point edit style: ps = PointStyle(Fixed, Color, DrawMode);
DrawMode - Defines the parametric point image type while editing:
DM_CIRCLE - circle(default);
DM_SQUARE - square;
DM_XARROW - x-arrows;
DM_YARROW - y-arrows;
Color - Defines the parametric point image color: integer value
If Z value is not specified it sets to 0.

Example of use:

```
P1 = ParameterPoint(Point(0, 0, 0));
P2 = ParameterPoint(0, 0, 0, 0);
bp1 = Point(0, 0);
bp2 = Point(1, 1);
P3 = ParameterPoint("First Point", 3.1415, 6.283, bp1, LOCK_Y, DM_XARROW);
P4 = ParameterPoint("Second Point", 3.1415, PointY(P3), DM_CIRCLE, 0xFF);
P5 = ParameterPoint("Third Point", 3.1415, 6.283, bp1, bp2, PointStyle(DM_CIRCLE, 0xFF));
```

Auxiliary Functions for Parameters and Parameter Points

Set

Sets the parameter value list.

Call format:

```
<id> = Set(<value1>, <value2>, ...);
<id> = Set(FolderList(<path>, <mask>));
```

Arguments:

value1, value2 - list of numeric values, separated with commas.
FolderList(path, mask) - files list returned by FolderList function

Example of using the function:

```
H = Parameter("Height", 36, LINEAR, Set(30, 36, 42));
```

FolderList

Creates a list of files with a certain name or extension stored in a specified folder.

Call format:

```
<id> = FolderList(<path>, <mask>);
```

Arguments:

path - path to the folder where the files are stored.
mask - file names or/and extension mask in format ["*.extension"].

Example of using the function:

```
Name = Parameter("File:", "test.dwg", Set(FolderList("<full path>", "*.dwg")));
```

Interval

Sets parameter restriction, so parameter value should fall within the specified interval.

Call format:
<id> = Interval(<minValue>, <maxValue>);

Arguments:

minValue - lower interval bound.
maxValue - upper interval bound.

Example of using the function:
D = Parameter("Depth", 14, LINEAR, Interval(10, 100));

LessThan

Sets parameter restriction, so parameter value should be less than the specified value.

Call format:
<id> = LessThan(<dValue>);

Arguments:

dValue - numeric value.

Example of using the function:
R = Parameter("Radius", 9, LINEAR, LessThan(10));

LessOrEqual

Sets parameter restriction, so parameter value should be less than or equal to the specified value.

Call format:
<id> = LessOrEqual(<dValue>);

Arguments:

dValue - numeric value.

Example of using the function:
p = Parameter("Radius", 9, LINEAR, LessOrEqual(10));

GreaterThan

Sets parameter restriction, so parameter value should be greater than the specified value.

Call format:
<id> = GreaterThan(<dValue>);

Arguments:

dValue - numeric value.

Example of using the function:
n = Parameter("Number", 10, LINEAR, GreaterThan(8));

GreaterOrEqual

Sets parameter restriction, so parameter value should be greater than or equal to the specified value.

Call format:
<id> = GreaterOrEqual(<dValue>);

Arguments:

dValue - numeric value.

PointStyle

Sets a point edit style including locking, color and draw mode.

Call format:

<id> = PointStyle(<list>);

Arguments:

list - properties separated with commas.
The following values of characteristics are available:
LOCK_X or LOCK_Y - coordinate locking (none - default).
integer value - color (magenta - default).
DM_CIRCLE – circle (default), DM_SQUARE - square, DM_XARROW - x-arrows or DM_YARROW - y-arrows - draw modes.

Example of using the function:

P5 = ParameterPoint("Third Point", 3.1415, 6.283, bp1, bp2, PointStyle(DM_CIRCLE, 0xFF));

Example of using the function:

n = Parameter("Number", 10, LINEAR, GreaterOrEqual(8));

Functions to Work with Points

Point

Creates a point.

Call format:

<id> = Point(<x>[, <y>, <z>]);

Arguments:

x - X coordinate of the point.
y - Y coordinate of the point. Default value is 0.
z - Z coordinate of the point. Default value is 0.

Example of using the function:

Pt1 = Point(1);
Pt2 = Point(1, 2);
Pt3 = Point(Pt1, Pt2); // X coordinate of Pt3 equals X coordinate of Pt1, Y coordinate of Pt3 equals Y coordinate of Pt2

PolarPoint

Creates a point defined in polar coordinates.

Call format:

<id> = PolarPoint(<length>, <angle>, <z>]);

Arguments:

length - distance between the point and Point(0,0).
angle - angle between the X- axis and the vector defined by the point.

Example of using the function:

Pt1 = PolarPoint(10, 45]);

PointX, PointY and PointZ

Returns the corresponding coordinate of a point.

Call format:

<id> = PointX(<point>);
<id> = PointY(<point >);
<id> = PointZ(<point >);

Arguments:

point – point.

Example of using the function:

```
Pt1 = Point(1);
Pt2 = Point(1, 2);
Pt3 = Point(PointY(Pt1), PointX(Pt2));
```

Function to Work with Graphic Styles and Properties

PPM language contains several ways to set draw styles and properties for graphic objects. The first of them is to define a property set and construct the graphic object using this property set. The second is to set properties for the graphic object directly, using SetProperties function.

PropertySet

Creates the property set that contains properties and can be used to create a graphic object.

Call format:

<ps> = PropertySet([<ps>,] <property name> = <property value>, <property name> = <property value>, …);

Arguments:

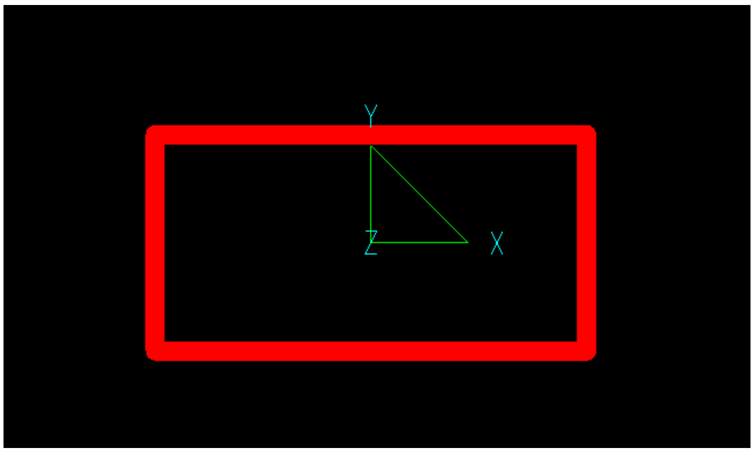
ps – base property set used for the new property set.

property name – defines the name of the property to be set. The name should be in quotation marks. The list of available properties is in the table below.

property value – defines the value of the property.

Example of using the function:

```
Red = PropertySet("Layer" = "Red", "PenColor" = 0x000000FF, "PenWidth" = 0);
Red2 = PropertySet(Red, "PenWidth" = 2);
Output(Rectangle(Red2, 10, 5));
```



Picture 4

SetProperties

Sets or resets the properties of graphic objects. This function is used for updating object properties such as Material, Pen Color and Brush Style.

Call format:

<gr> = SetProperies(<gObject>, <property name> = <property value>, <property name> = <property value>, …);

Arguments:

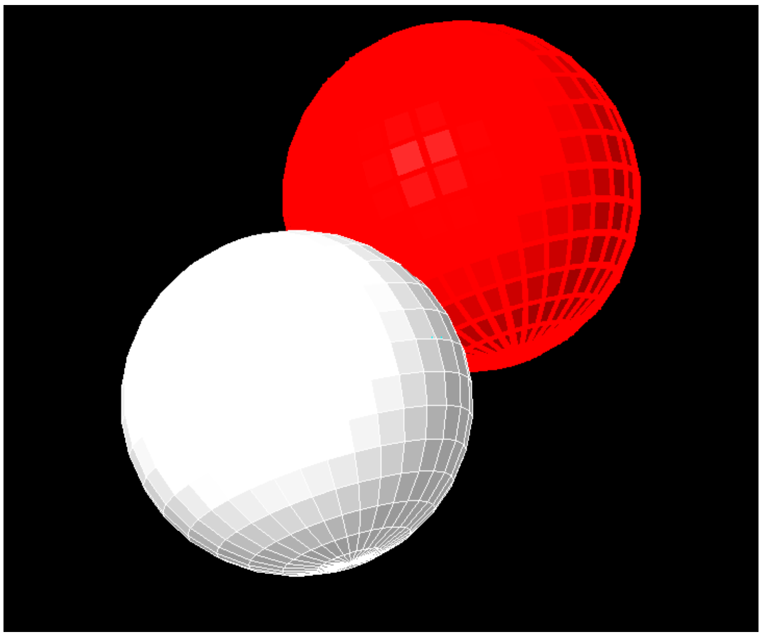
gObject – object to set properties to.

property name – defines the name of the property to be set. The name should be in quotation marks. The list of available properties is in the table below.

property value – defines the value of the property.

Example of using the function:

```
s = Sphere(2);
id = SetProperties(Move(s, Point(2, 2, 2)), "Layer" = "Red", "PenColor" = 0x000000FF, "PenWidth" = 0.5);
Output(id, s);
```



Picture 5

Object Type	Property Name	Value Type	Description
	PenColor	Integer	Sets the color of the graphic object.
	PenWidth	Real	Sets the line weight of the graphic object.
	PenStyle	String(Name of line type)	Sets the line type of the graphic object.
	PenScale	Real	Sets the line type scale factor of the graphic object.
	Thickness	Real	Sets the thickness the graphic object. Note: Only for Polyline, Circle and Arc.
	Layer	String(Name of layer)	Sets the layer for the graphic object.
	Material	String(Name of material)	Sets the material for the graphic object.
	BrushStyle	String(Name of pattern)	Sets the pattern for hatch for the graphic object.
	BrushScale	Real	Sets the scale of the hatch for the graphic object.
	BrushColor	Integer	Sets the color of the hatch for the graphic object.
	BrushAngle	Real	Sets the angle of the hatch for the graphic object.
Graphic	BrushTransp	Integer(0-100)	Sets the transparency of hatch for the graphic object.
	TFont	String(Font name)	Sets the font for the text.
	THeight	Real	Sets the height of the text.
	TWidthf	Real	Sets the width factor for the text.
Text	TOblA	Real	Sets angle (in radians) to be the obliquing angle for the text.

	TAlign	Combination of alignment borders. {TOP, CENTER}	Sets the justification of the text. Possible values are LEFT, CENTER, RIGHT, TOP, MIDDLE, BASELINE, BOTTOM.
	TStyle	Combination of text style flags. {BOLD, ITALIC}	Sets the text style properties. Possible values are BOLD, ITALIC and UNDERLINE.
Dimension	DimTxt	Real	Sets the height for the text of the dimension.
	DimScale	Real	Sets the scale factor for the text of the dimension.
	DimAsz	Real	Sets the arrowhead size for dimensions.
	DimClrd	Integer	Sets the color of dimension lines, leader lines, frames, and arrowheads.
	DimClre	Integer	Sets the color of extension lines.
	DimClrt	Integer	Sets the color of dimension text
	DimTh	Boolean	0 or 1. Sets the orientation of dimension text inside/outside the extension lines.
	DimDec	Integer	Sets the number of decimal places in primary units in dimensions.
	DimADec	Integer	Sets the number of decimal places in angular dimensions.
Image	ImgBrightness	Integer	Sets the brightness of the image. Not Supported in AutoCAD.
	ImgContrast	Integer	Sets the contrast of the image. Not Supported in AutoCAD.
	ImgFade	Integer	Sets the fade of the image. Not Supported in AutoCAD.
	ImgTransp	Real	Sets the transparency of the image. Not Supported in AutoCAD.

Functions for creating 2D primitives

Arc

Creates a single arc.

Call format:

```
<id> = Arc([<PS>, ]<dRadius>, <aStart>, <aEnd>, <pCenter>);
<id> = Arc([<PS>, ]<dRadius>, <aStart>, <aEnd>, <dCenterX>, <dCenterY>);
<id> = Arc([<PS>, ]<dRadius>, <pStart>, <pEnd>, <pCenter>);
<id> = Arc([<PS>, ]<dRadius>, <pStart>, <pEnd>, <dCenterX>, <dCenterY>);
<id> = Arc([<PS>, ]<pStart>, <pBetween>, <pEnd>);
```

Arguments:

PS - Property Style (optional parameter).
dRadius - arc radius value.
aStart - start building angle.
aEnd - end building angle.
pStart - arc start point.
pEnd - arc endpoint .
pCenter - arc center point defined by coordinates.
pBetween - a point on the arc circumference;
dCenterX - arc center X-coordinate.
dCenterY - arc center Y-coordinate.

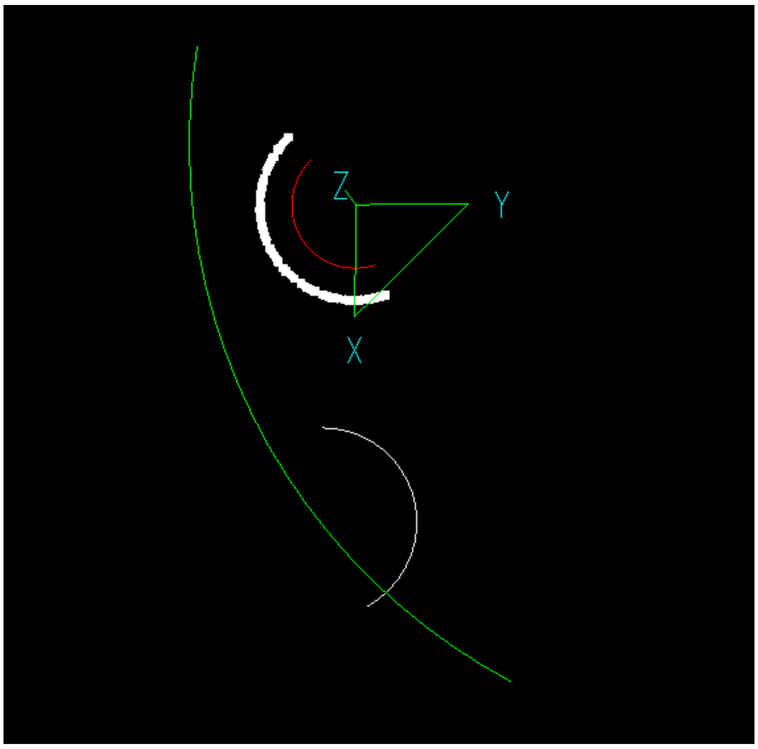
Example of using the function:

```
//***Property Sets***
Red = PropertySet("PenColor" = 0x000000FF);
Green = PropertySet("PenColor" = 0x0000FF00, "PenWidth" = 0);
//***End***
```

```
Radius = 3;
aStart = 30; aEnd = 180;
pCenter = Point(10,-1,0);
pStart = Point(-5,-5);
pEnd = Point(15,5);
pBetween = pCenter;
id1 = Arc(Radius, aStart, aEnd, pCenter);
Output(id1);

id2 = Arc(Red, Radius-1, pStart, pEnd);
id3 = Arc(PropertySet("PenStyle" = "DASHED", "PenWidth" = 0.5), Radius, pStart, pEnd,0, 0);
Output(id2, id3);

id4 = Arc(Green, pStart, pBetween, pEnd);
Output(id4);
```



Picture 6

Bezier

Creates a Bezier curve.

Call format:

```
<id> = Bezier([<PS>, ]<p1>, <q1>, <q2>, <p2>);
<id> = Bezier([<PS>, ]<p1>, <q1>, <q2a>, <p2>[, <q2b>, <q3a>, <p3>]...);
<id> = Bezier([<PS>, ]<sPolyline>);
```

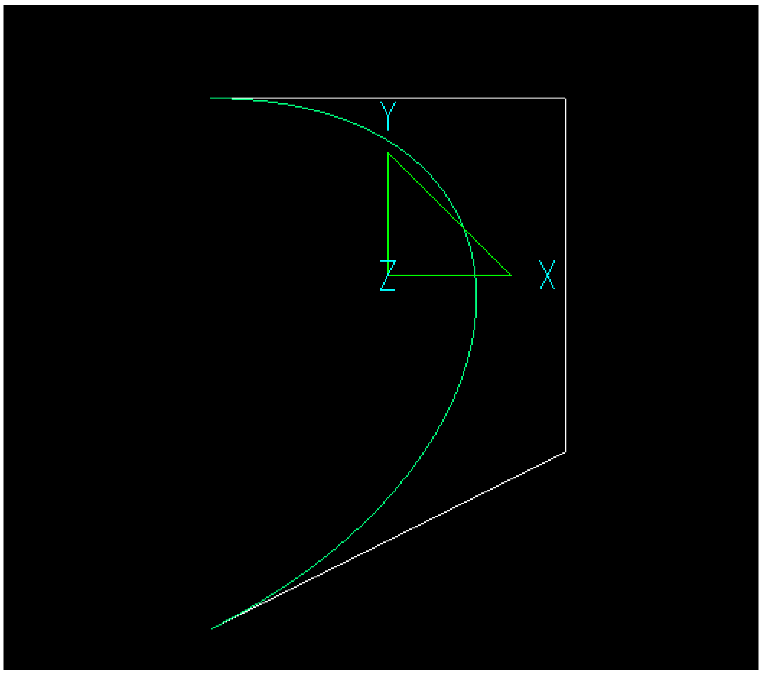
Arguments:

PS - Property Style (optional parameter).
sPolyline - polyline to be converted.

Example of using the function:

```
//***Property Sets***
Green = PropertySet("PenColor" = 0x007FFF00);
```

```
//***End***
poly = Polyline(Point(-1, 1), Point(1, 1), Point(1, -1), Point(-1, -2));
b1 = Bezier(Green, poly);
Output(poly, b1);
```



Picture 7

Circle

Creates a Circle object.

Call format:

```
<id> = Circle([<PS>, ]<dRadius>, <dCenterX>, <dCenterY>);
<id> = Circle([<PS>, ]<dRadius>);
<id> = Circle([<PS>, ]<dRadius>, <pCenter>);
<id> = Circle([<PS>, ]<pCenter>, <pRadius>);
<id> = Circle([<PS>, ]<point1>, <point2>, <point3>);
<id> = Circle([<PS>, ]<point1>, <point2>, <point2>); // by 2 points
```

Arguments:

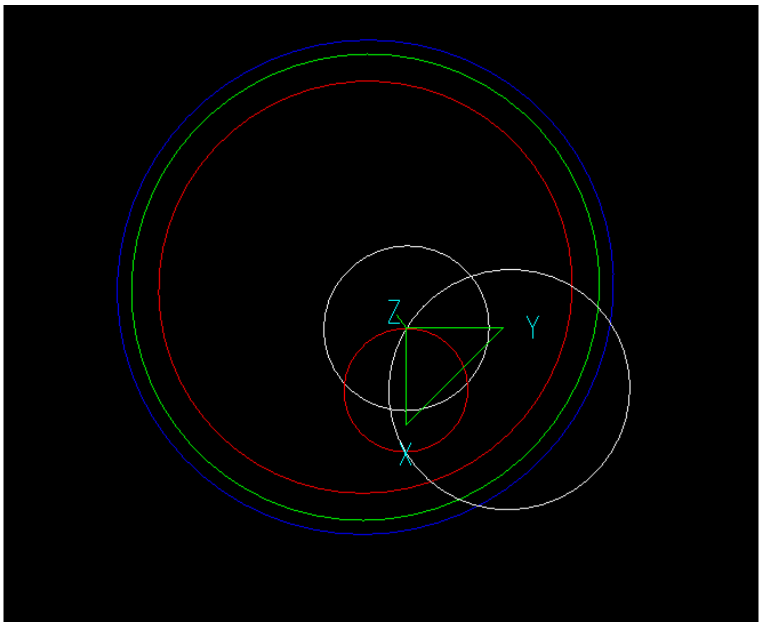
PS - Property Style (optional parameter)
dRadius - radius value.
dCenterX - X-coordinate value of the circle center.
dCenterY - Y-coordinate value of the circle center.
Note: When the coordinate values are not specified, their values are set to "0".
pCenter - circle center point defined by X and Y coordinates.
pRadius - circle radius endpoint defined by X and Y coordinates.
point1 - point on the circumference defined by X and Y coordinates.
point2 - point on the circumference defined by X and Y coordinates.
point3 - point on the circumference defined by X and Y coordinates.

Example of using the function:

```
//***Property Sets***
Red = PropertySet("Layer" = "Red", "PenColor" = 0x000000FF, "PenWidth" = 0);
Green = PropertySet("Layer" = "Green", "PenColor" = 0x0000FF00, "PenWidth" = 0);
```

```
Blue = PropertySet("Layer" = "Blue", "PenColor" = 0x00FF0000, "PenWidth" = 0);
//***End***

dRadius = 5;
dCenterX = -1; dCenterY = -1;
pCenter = Point(dCenterX, dCenterY);
pRadius = Point (3, 3);
point1 = Point (0, 0);
point2 = Point (3, 0);
point3 = Point (4, 4);
id1 = Circle(Blue, dRadius+1, dCenterX, dCenterY);
id2 = Circle(dRadius-3);
id3 = Circle(Red, dRadius, pCenter);
id4 = Circle(Green, pCenter, pRadius);
id5 = Circle(point1, point2, point3);
id6 = Circle(Red, point1, point2, point2);
Output (id1, id2, id3, id4, id5, id6);
```



Picture 8

Ellipse

Creates an Ellipse object.

Call format:

```
<id> = Ellipse([<PS>, ]<xSize>, <ySize>, <dCenterX>, <dCenterY>);
<id> = Ellipse([<PS>, ]<xSize>, <ySize>, <pCenter>);
<id> = Ellipse([<PS>, ]<xSize>, <ySize>);
<id> = Ellipse([<PS>, ]<point1>, <point2>);
```

Arguments:

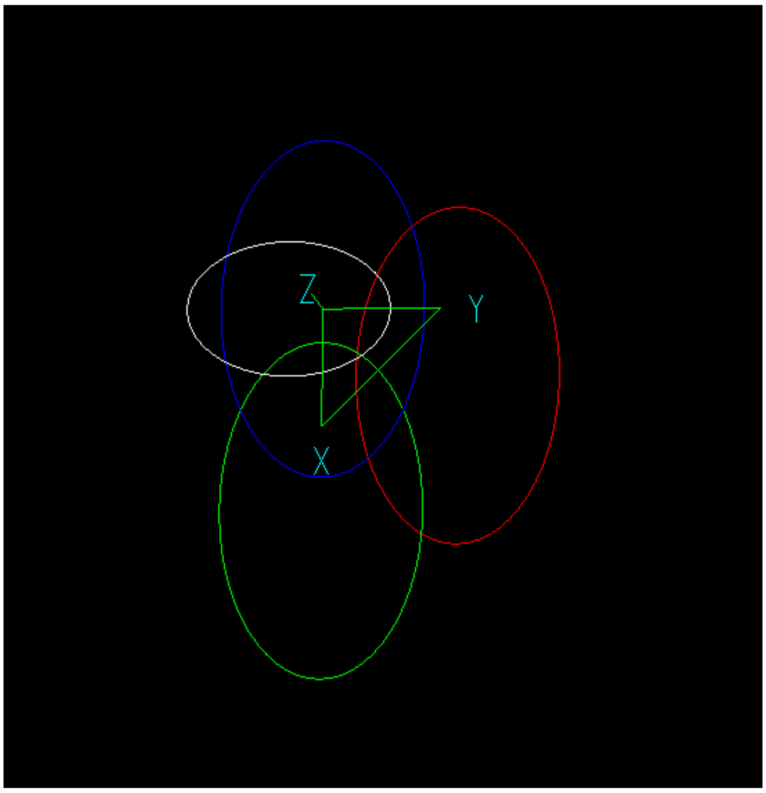
PS - Property Style (optional parameter)
xSize - horizontal axis length value.
ySize - vertical axis length value.
dCenterX - X-coordinate value of the bounding rectangle center. When the value is not specified, it defaults to 0.

dCenterY - Y-coordinate value of the bounding rectangle center. When the value is not specified, it defaults to 0.
pCenter - the center point of the bounding rectangle center defined by X and Y coordinates.
point1 - the first of two diagonally opposite vertexes of the bounding rectangle defined by X and Y coordinates.
point2 - the second of two diagonally opposite vertexes of the bounding rectangle defined by X and Y coordinates.

Example of using the function:

```
//***Property Sets***
Red = PropertySet("Layer" = "Red", "PenColor" = 0x000000FF, "PenWidth" = 0);
Green = PropertySet("Layer" = "Green", "PenColor" = 0x0000FF00, "PenWidth" = 0);
Blue = PropertySet("Layer" = "Blue", "PenColor" = 0x00FF0000, "PenWidth" = 0);
//***End***

dWidth = 5; dHeight= 3;
point1 = Point(1, 1); point2 = Point(-1, -2);
dCenterX = PointX(point1); dCenterY = 2;
//example of using function Ellipse
xSize = dWidth; ySize = dHeight;
pCenter = Point(3, 0);
el1 = Ellipse(Red, xSize, ySize, dCenterX, dCenterY);
el2 = Ellipse(Green, xSize, ySize, pCenter);
el3 = Ellipse(Blue, xSize, ySize);
el4 = Ellipse(point1, point2);
Output(el1, el2, el3, el4);
```



Picture 9

Rectangle

Creates a Rectangle object.

Call format:

```
<id> = Rectangle([<PS>, ]<dWidth>, <dHeight>, <dCenterX>, <dCenterY>);
<id> = Rectangle([<PS>, ]<dWidth>, <dHeight>);
<id> = Rectangle([<PS>, ]<point1>, <point2>);
```

Arguments:

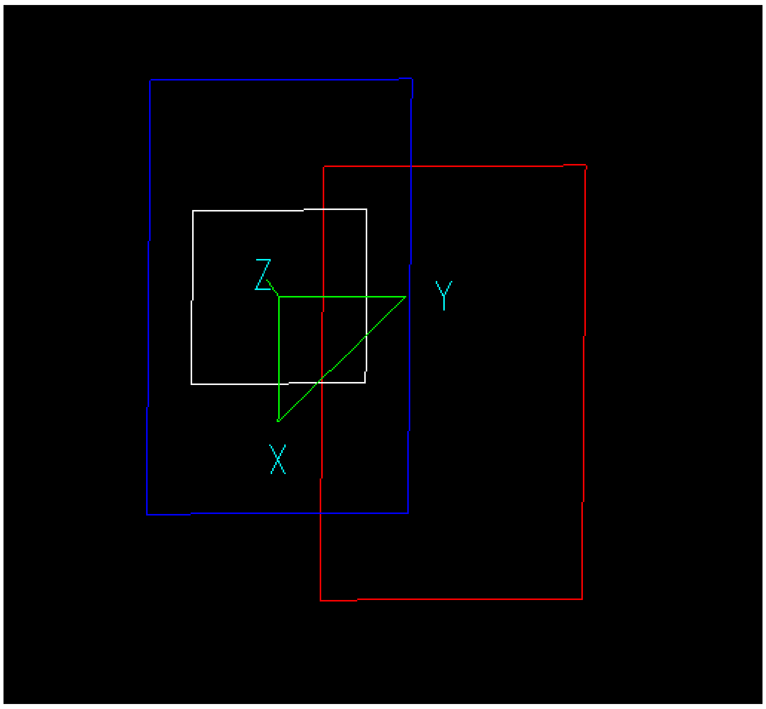
PS - Property Style (optional parameter)
dWidth - Numeric value of the rectangle width.
dHeight - Numeric value of the rectangle height.
dCenterX - Numeric value coordinates of the rectangle center. The default value is set to "0".
dCenterY - Numeric value coordinates of the rectangle center. The default value is set to "0".
point1 - the first of two diagonally opposite vertexes of the bounding rectangle defined by X and Y coordinates.
point2 - the second of two diagonally opposite vertexes of the bounding rectangle defined by X and Y coordinates.

Example of using the function:

```
//***Property Sets***
Red = PropertySet("Layer" = "Red", "PenColor" = 0x000000FF, "PenWidth" = 0);
Blue = PropertySet("Layer" = "Blue", "PenColor" = 0x00FF0000, "PenWidth" = 0);
//***End***

dWidth = 5; dHeight = 3;
point1 = Point(1,1); point2 = Point(-1,-1);
dCenterX = PointX(point1); dCenterY = 2;

id1 = Rectangle(Red, dWidth, dHeight, dCenterX, dCenterY);
id2 = Rectangle(Blue, dWidth, dHeight);
id3 = Rectangle(point1, point2);
Output(id1, id2, id3);
```



Picture 10

Polyline

Creates 2D Polyline object that may include straight line segments and arc segments. A line segment is defined by two points. Bulge, Fillet, Arc0 and Arc1 functions define arc segments of the polyline. The polyline where the first and last points are coincident is called a closed polyline. Closed polyline can be used for creating 3D objects.

Call format:

<id> = Polyline([<PS>,]<list of arguments>);

Arguments:

PS - Property Style (optional parameter)

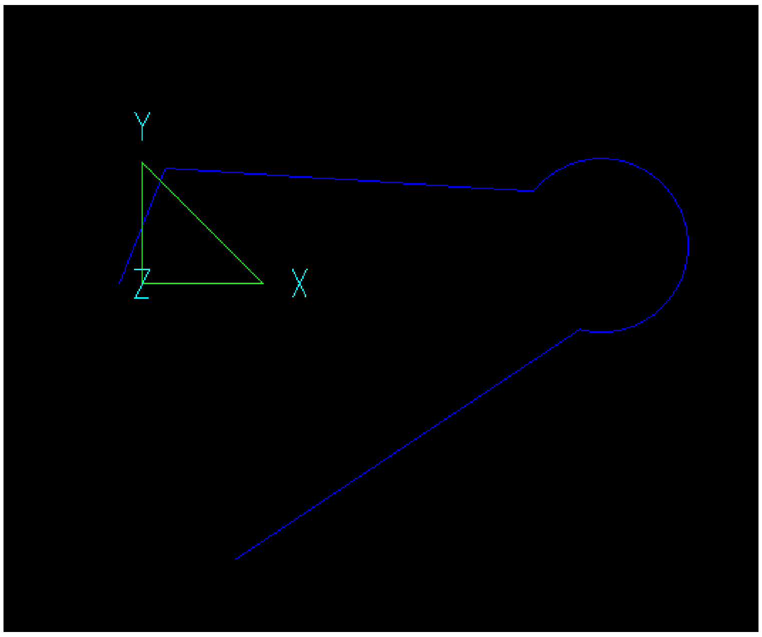
list of arguments - Defines the list of arguments, separated with commas. Arguments define individual segments of a polyline.

Example of using the function:

```
//***Property Sets***
Blue = PropertySet("Layer" = "Blue", "PenColor" = 0x00FF0000, "PenWidth" = 0);
//***End***

p1 = Point(-10, 0, 0);
x = 175; y = 30;
p3 = Point(190, -20, 0);
p4 = Point(40, -120, 0);

g = Polyline(Blue, p1, Point(10, 50, 0), Point(170, 40, 0), Arc0(x, y, 3), p3, p4);
Output(g);
```



Picture 11

PolyBezier

Creates a set of Bezier curves.

Call format:

<id> = PolyBezier([<PS>,]<p1>, <p2>, ...);

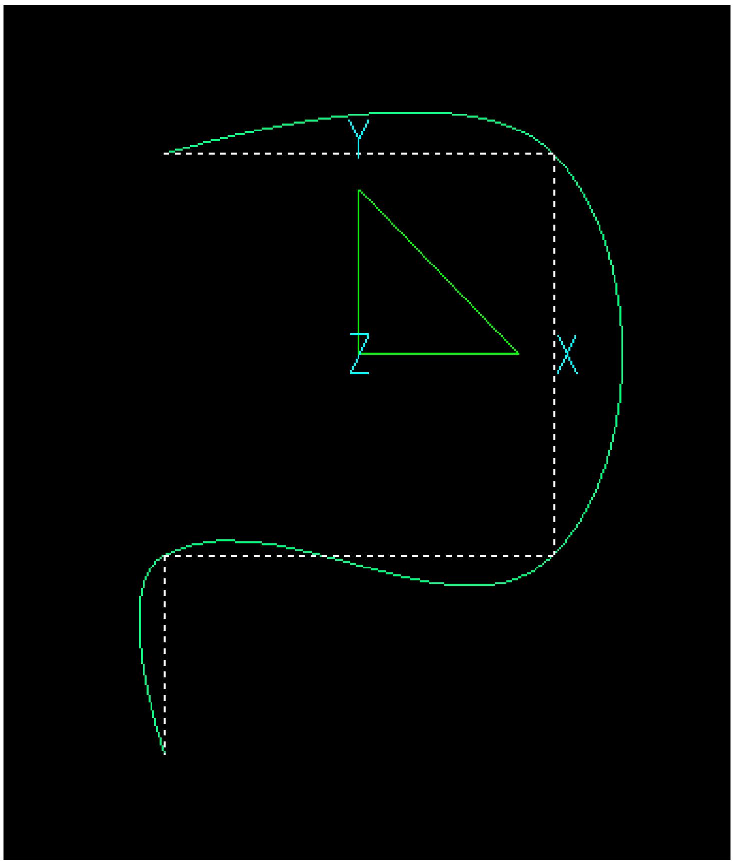
<id> = PolyBezier([<PS>,]<sPolyline>);

Arguments:

PS - Property Style (optional parameter)
p1, p2 – the list of control points of the Bezier curve.
sPolyline - polyline to be converted.

Example of using the function:

```
//***Property Sets***
Green = PropertySet("PenColor" = 0x007FFF00);
//***End***
poly = Polyline(Point(-1, 1), Point(1, 1), Point(1, -1), Point(-1, -1), Point(-1, -2));
b1 = PolyBezier(Green, poly);
Output(poly, b1);
```



Picture 12

Auxiliary Functions for Polyline

Polylines with arc segments are defined by adding auxiliary functions *Arc0* and *Arc1* to the list of arguments. Arc0 builds the circular arc clockwise, while Arc1 builds the circular arc counterclockwise. Fillet function rounds two adjacent polyline segments.

Arc0

Creates a clockwise arc segment in a polyline. The start and end points of an arc are defined by the preceding and succeeding points in the list of the Polyline function arguments. So, Arc0 cannot be the first or the last argument of the Polyline function.

Call format:

```
<id> = Arc0(<dRadius>);
<id> = Arc0(<x>, <y>);
<id> = Arc0(<pCenter>);
```

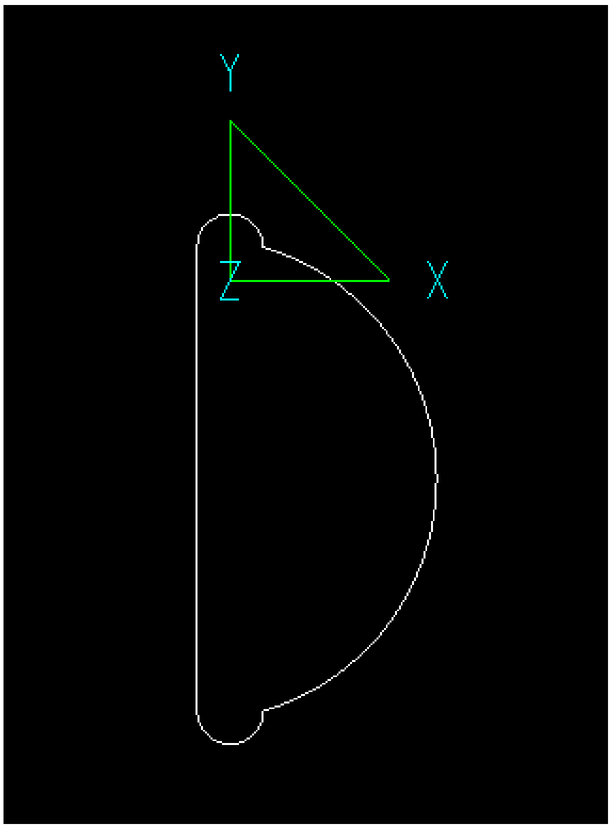
Arguments:

x - X-coordinate of arc center point.
y - Y-coordinate of arc center point.
dRadius - arc radius value.
pCenter - arc center point.

Example of using the function:

```
R2 = 10;
l1 = 120;
p#1 = Point(-R2, -(l1 + R2));
p#2 = Point(-R2, -(l1 + R2)/2);
p4 = Polyline(p#1, Point(-R2, R2),
                   Arc0(R2),
                   Point(R2, R2),
                   Arc0(p#2),
                   Point(R2, -(l1 + R2)),
                   Arc0(0, -l1 - R2), p#1);

Output(p4);
```



Picture 13

Arc1

Creates a counterclockwise arc segment in a polyline. The start and end points of an arc are defined by the preceding and succeeding points in the list of the Polyline function arguments. So, Arc1 cannot be the first or the last argument of the Polyline function.

Call format:

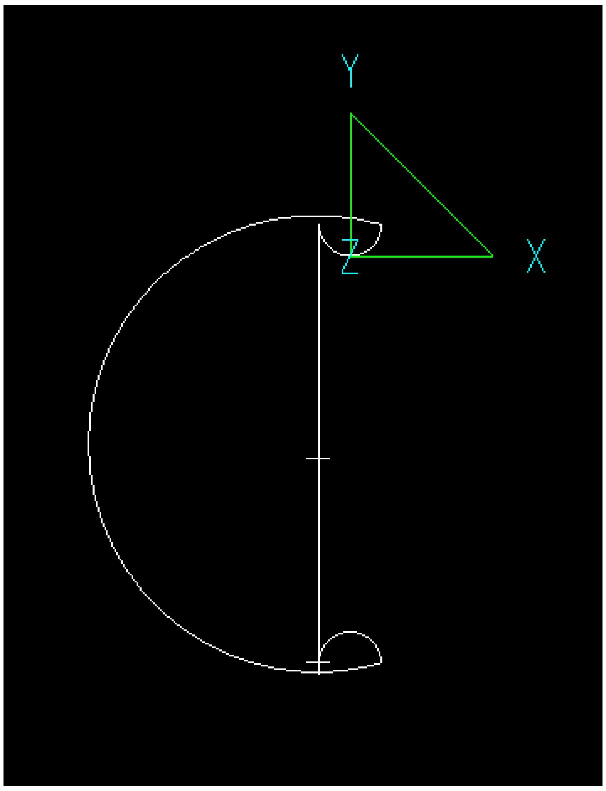
```
<id> = Arc1(<dRadius>);
<id> = Arc1(<x>, <y>);
<id> = Arc1(<pCenter>);
```

Arguments:

x - X-coordinate of arc center point.
y - Y-coordinate of arc center point.
dRadius - arc radius value.
pCenter - arc center point.

Example of using the function:

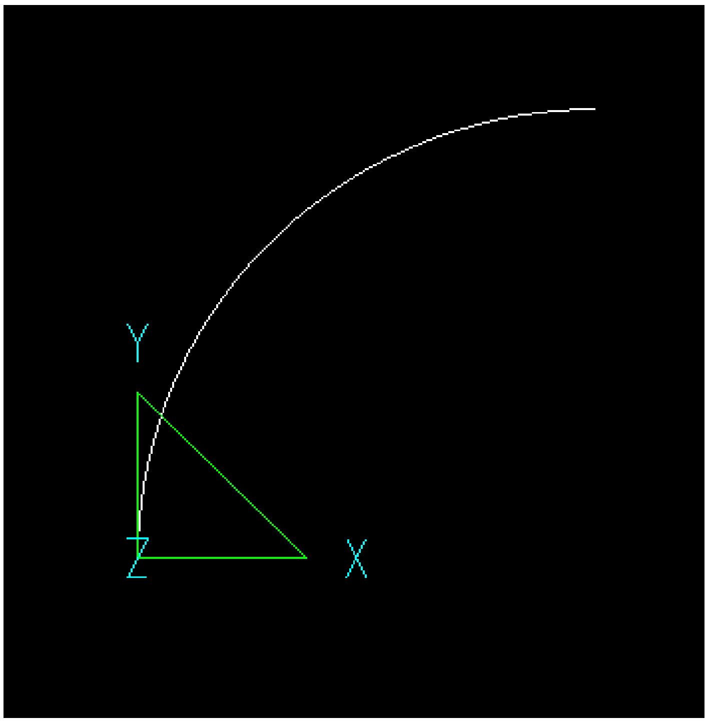
```
R2 = 10;
l1 = 120;
p#1 = Point(-R2, -(l1 + R2));
p#2 = Point(-R2,-(l1 +  R2)/2);
p4 = Polyline(p#1, Point(-R2, R2),
                           Arc1(R2),
                           Point(R2, R2),
                           Arc1(p#2),
                           Point(R2, -(l1 + R2)),
                           Arc1(0,-l1 - R2), p#1);
Output(p4);
```



Picture 14

Note: To define a polyline that contains only an arc segment, the list of arguments should contain the start point of the arc, Arc1 or Arc0 and the end point of the arc.

```
Output(Polyline(Point(0, 0), Arc0(1), Point(1, 1)));
```



Picture 15

Bulge

Defines the current Polyline segment as an arc segment with the specified bulge value. Bulge value is the ratio between the arc height and the half of the chord length. Bulge cannot be the first or the last argument of the Polyline function.

Call format:

```
<id> = Bulge(dBulge);
```

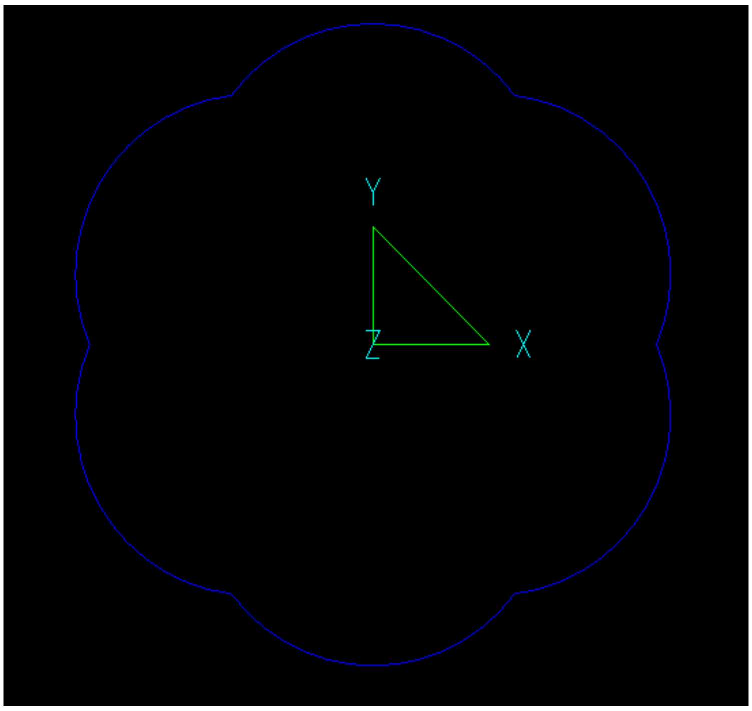
Arguments:

dBulge - bulge value for created arc.

Example of using the function:

```
//***Property Sets***
Blue = PropertySet("Layer" = "Blue", "PenColor" = 0x00FF0000, "PenWidth" = 0);
//***End***

b=0.5;
pp = Polyline(Blue,
                Point(10,0), Bulge(b),
                PolarPoint(10, 60[deg]), Bulge(b),
                PolarPoint(10, 120[deg]), Bulge(b),
                PolarPoint(10, 180[deg]), Bulge(b),
                PolarPoint(10, 240[deg]), Bulge(b),
                PolarPoint(10, 300[deg]), Bulge(b),
                PolarPoint(10, 360[deg]));
Output(pp);
```



Picture 16

Fillet

Rounds two adjacent polyline segments. This function can be used as a parameter of the Polyline and the Polyline3D functions. Fillet cannot be the first or the last argument.

Call format:

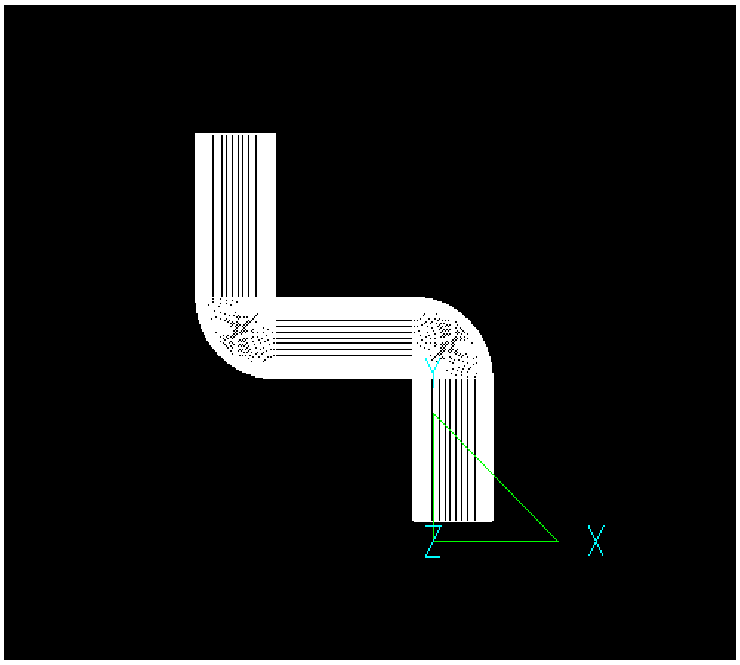
```
<id> = Fillet(<dRadius>);
```

Arguments:

dRadius - Fillet arc radius value.

Example of using the function:

```
P1 = Point(1, 10);
P2 = Point(-10, 10);
R = 2;
id = Polyline(Point(1, 1), P1, Fillet(R), P2, Fillet(R), Point(-10, 20));
s = Sweep(Circle(2), id, 0);
Output(s, id);
```



Picture 17

Offset

Creates the next polyline point by shifting the previous one by the specified value.

Call format:

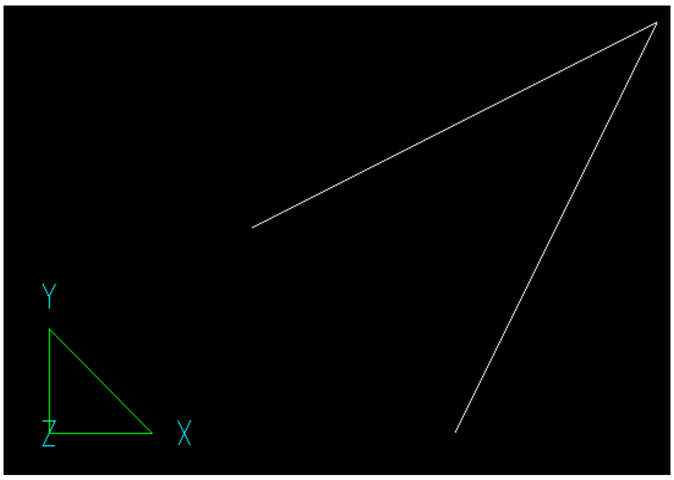
<id> = Offset(<pOffset>);
<id> = Offset(<dx>, <dy>);

Arguments:

dx, dy - coordinates.
pOffset - point which defines the offset.

Example of using the function:

pol = Polyline(Point(1, 1), Offset(2, 1), Offset(Point(-1, -2)));
Output(pol);



Picture 18

PolarOffset

Creates the next polyline point, based on a previous segment and the offset value in Polar coordinates.

Call format:

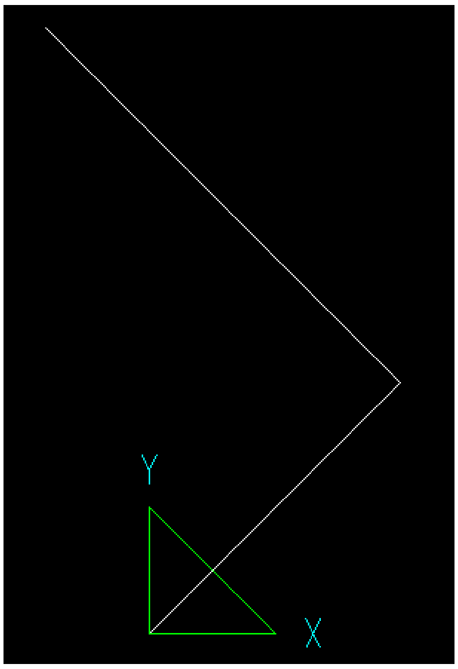
<id> = PolarOffset(<dLength>, <dAngle>);

Arguments:

dLength - distance to the created polyline point from the previous one.
dAndgle - angle between the previous polyline segment and new one.

Example of using the function:

```
p0 = Point(0, 0);
p1 = Point(5, 5);
pol = Polyline(p0, p1, PolarOffset(10, 90[deg]));
Output(pol);
```



Picture 19

Functions for Creating Text

Text

Creates graphical text.

Call format:

<id> = Text([<PS>,]<tText>, <tFont>, <tStyle>);
<id> = Text([<PS>,]<tText>, <p1>, <tRot>)

Arguments:

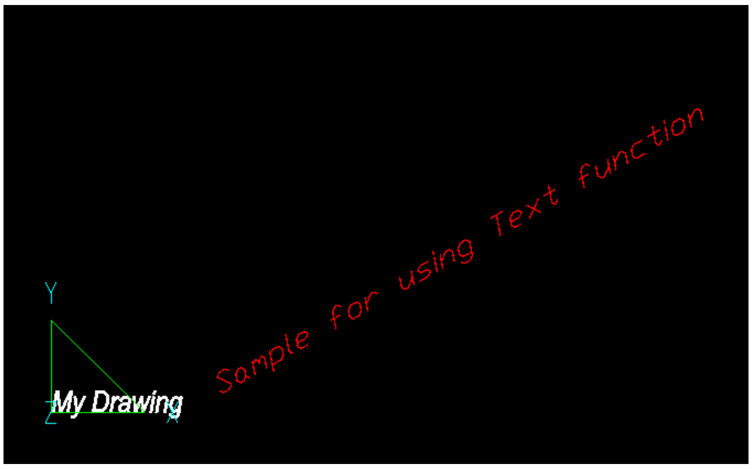
PS - Property Style (optional parameter)
tText - text string.
tFont - call of the function TextFont (see description)
tStyle - call of the function TextStyle (see description)
p1 - point defined the text position.
tRot - text rotation angle.

Example of using the function:

```
//***Property Sets***
RedItalicBox = PropertySet("PenColor" = 0x000000FF, "THeight" = 12, "TStyle" = {ITALIC,BOX});
//***End***
```

```
t_Font = TextFont(0, 20, 0, "Arial Narrow");
t_Style = TextStyle({ITALIC, BOX});
id = Text("My Drawing", t_Font, t_Style);
id2 = Text(RedItalicBox, "Sample for using Text function", Point(100, 10), 30);

Output(id2, id);
```



Picture 20

TextFont

Sets text font, size and angle of the text line position. This is an auxiliary function.

Call format:

<id> = TextFont(<tMode>, <tHeight>, <tAngle>, <tFont>);

Arguments:

tMode - text mode: Standard (when mode=0) or Scalable (when mode=1 or any value other than "0").

tHeight - text font size.

tAngle - text line position angle.

tFont - one of a text fonts specified as ["Font"].

Example of using the function:

See example for Text function

TextStyle

Sets text style including justification, text effects and styles.

Call format:

<Id> = TextStyle(<list>);

Arguments:

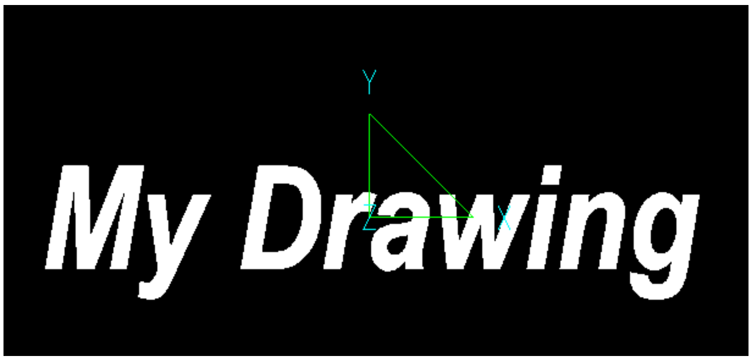
list - text properties separated with commas.

The following values of characteristics are available:

- LEFT, CENTER, RIGHT, TOP, MIDDLE, BASELINE, BOTTOM - Justification properties.
- BOLD, ITALIC, UNDERLINE - Style properties.

Example of using the function:

```
t_Font = TextFont(0, 30, 0, "Arial Narrow");
t_Style = TextStyle({UNDERLINE, ITALIC, BOLD, MIDDLE, CENTER});
id = Text("My Drawing", t_Font, t_Style);
Output(id);
```



Picture 21

Functions for Calculations and Geometrical Operations

AngleXY

Calculates the angle between the X-axis and the line defined by two points.

Call format:

```
<id> = AngleXY(<Point1>, <Point2>);
```

Arguments:

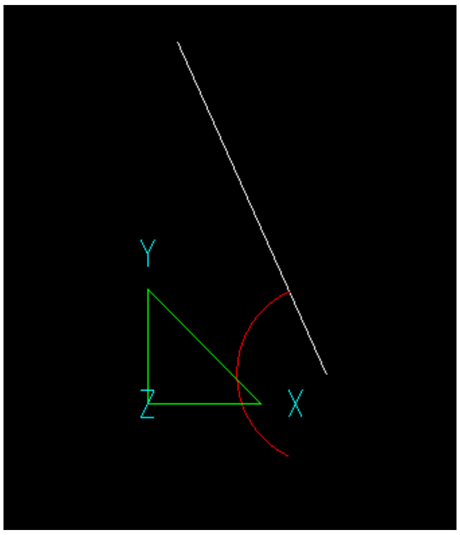
Point1 - the first point defined by coordinates.
Point2 - the second point defined by coordinates.

Example of using the function:

```
//***Property Sets***
Red = PropertySet("PenColor" = 0x000000FF);
//***End***

Point1 = ParameterPoint("Point1", 6, 1);
Point2 = ParameterPoint("Point2", 1, 12);
alfa = AngleXY(Point1, Point2);

line1 = Polyline(Point1, Point2);
R = 3;
a = Arc(Red, R, alfa, -alfa, Point1)
Output(line1, a);
```



Picture 22

CenterPoint

Returns center point for arcs and circles. For other graphical objects the function returns the center of extents.

Call format:

```
<id> = CenterPoint(<gObject>);
```

Arguments:

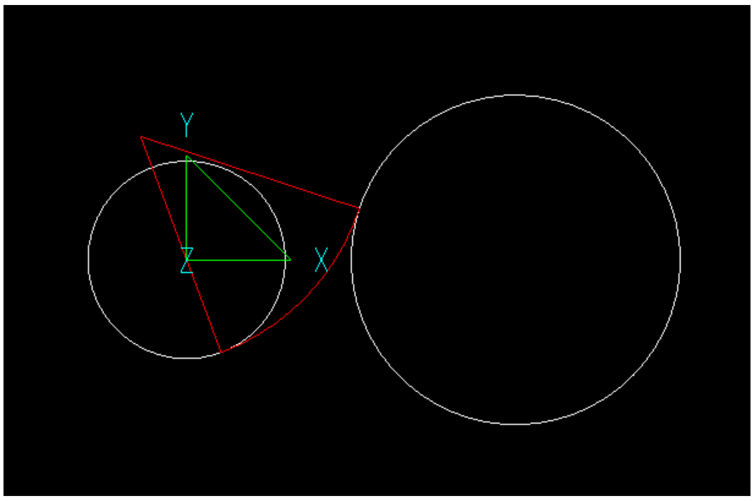
gObject - graphical Object.

Example of using the function:

```
//***Property Sets***
Red = PropertySet("Layer" = "Red", "PenColor" = 0x000000FF);
//***End***

R = 7;
c1 = Circle(3);
c2 = Circle(5, Point(10, 0));
Output(c1, c2);

arc = TangentArc(R, c1, c2, Point(0, 0), Point(10, -10));
Output(arc);
p#1 = CenterPoint(arc);
p1 = Polyline(Red, p#1, arc[1], Arc0(R),  arc[0], p#1);
Output(p1);
```



Picture 23

Chamfer (Not Implemented)

Connects two lines (single or double lines) with a beveled corner.

Call format:

```
<id> = Chamfer(<gObject>, <dRadius>, <pCenter>);
```

Arguments:

gObject - graphical object defined by name or an expression.
dRadius - numeric value of chamfer radius.
pCenter – point with coordinates chamfer center arc.

Example of using the function:

CircleXCircle

Calculates intersection of two circles.

Call format:

<id> = CircleXCircle(<pCenter1>, <R1>, <pCenter2>, <R2>, <pTo>);

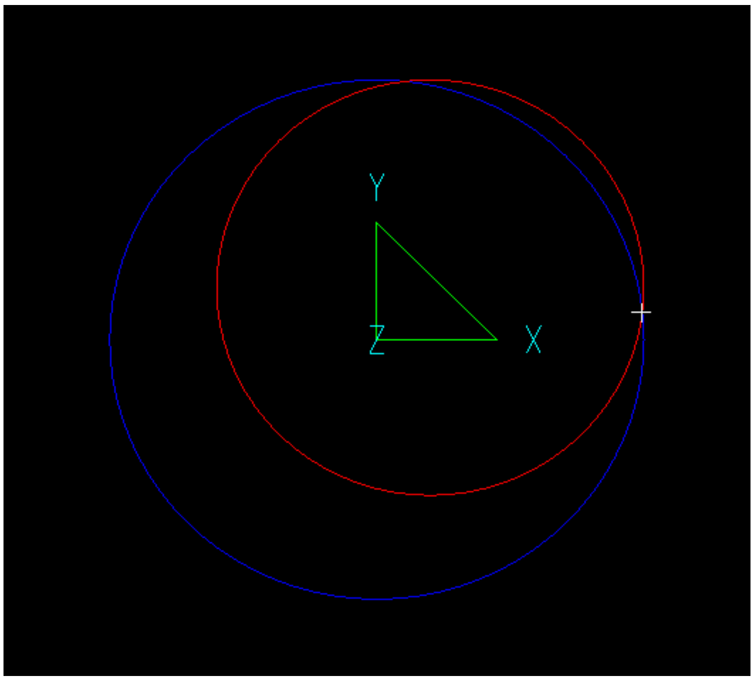
Arguments:

pCenter1, pCenter2 - center points of the circles.
R1, R2 - radiuses of the circles.
pTo - point to make choice between variants.

Example of using the function:

```
//***Property Sets***
Red = PropertySet("Layer" = "Red", "PenColor" = 0x000000FF);
Green = PropertySet("Layer" = "Green", "PenColor" = 0x0000FF00);
Blue = PropertySet("Layer" = "Blue", "PenColor" = 0x00FF0000);
//***End***

c1 = Circle(Blue, 5);
c2 = Circle(Red, 4 , Point(1, 1));
Output(c1, c2);
pTo = Point(100, 100);
pResult = CircleXCircle(Point(0, 0), 5, Point(1, 1), 4, pTo);
Output(pResult);
```



Picture 24

ClipPolyline

Clips polyline.

Call format:

<id> = ClipPolyline([<PS>,]<gPolyline>, <pTo>);
<id> = ClipPolyline([<PS>,]<gPolyline, <pFrom>, <pTo>);

Arguments:

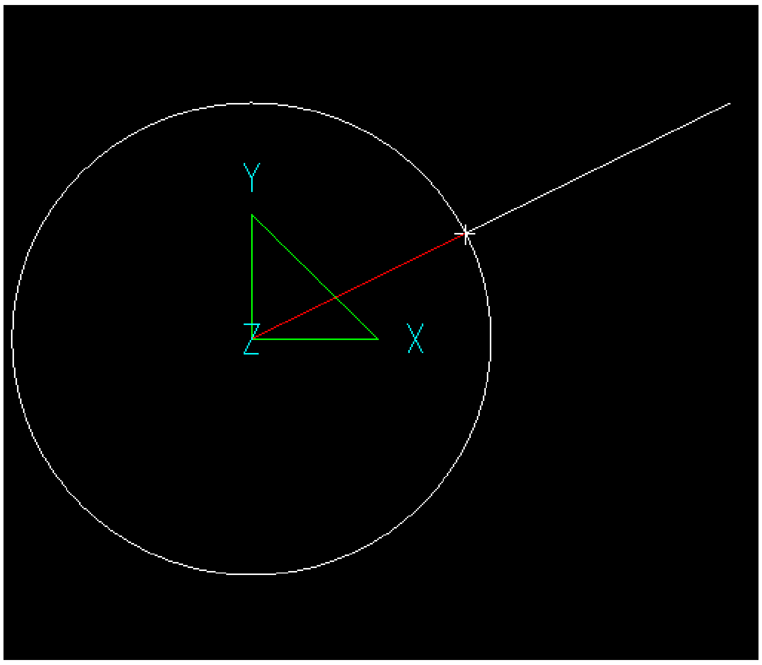
PS - Property Style (optional parameter).

gPolyline - source polyline.
pFrom - first point for clipping.
pTo - second point for clipping.

Example of using the function:

```
//***Property Sets***
Red = PropertySet("Layer" = "Red", "PenColor" = 0x000000FF);
//***End***
R = 1;
c1 = Circle(R);
p1 = Polyline(Point(0, 0), Point(2, 1));
Output(c1, p1, pInt);
pTo = Point(0, 0);
pInt = LineXCircle(Point(0, 0), Point(2, 1), Point(0,0), R, pTo);

CP = ClipPolyline(Red, p1, pInt);
Output(CP);
```



Picture 25

CutAndTransform

Cuts a polyline or array and rotates the left part.

Call format:

```
<id> = CutAndTransform([<PS>, ]<Array>, <p1>, <p2>, <angle>)
<id> = CutAndTransform([<PS>, ]<Array>, <p1>, <p2>, <m>)
<id> = CutAndTransform([<PS>, ]<poly1>, <p1>, <p2>, <angle>)
<id> = CutAndTransform([<PS>, ]<poly1>, <p1>, <p2>, <m>)
```

Arguments:

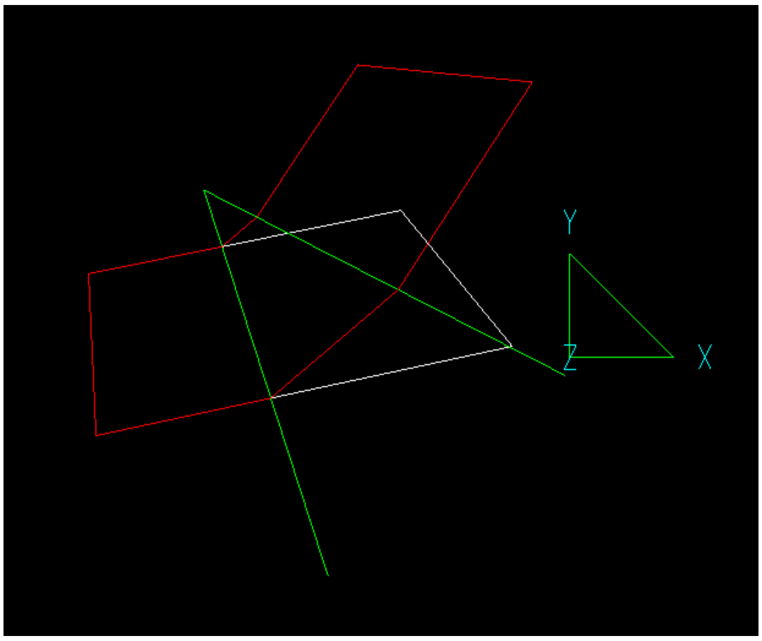
PS - Property Style (optional parameter).
Arrray – source array.
gPolyline - source polyline.
p1, p2 - points that define the cut line.
angle - rotation angle.
m - transformation matrix (i.e. rotation of the left part around p1).

Example of using the function:

```
//***Property Sets***
Red = PropertySet("Layer" = "Red", "PenColor" = 0x000000FF);
Green = PropertySet("Layer" = "Green", "PenColor" = 0x0000FF00);
//***End***

Angle = Parameter("Angle", 45, ANGULAR, Set(30, 45, 60, 90));
p1#2 = Point(-2.55, -0.42, 0);
p2#2_1 = Point(-2.59, 0.45, 0);
p2#2_2 = Point(-0.91, 0.79, 0);
p2#2_3 = Point(-0.31, 0.06, 0);
p2#2_5 = Point(-2.55, -0.42, 0);
p2#2 = Array(p2#2_1, p2#2_2, p2#2_3, p2#2_5);
g#2 = Polyline(p1#2, p2#2);
p1#3 = Point(-1.97, 0.9, 0);
p2#3 = Point(-1.3, -1.18, 0);
g#3 = Polyline(Green, p1#3, p2#3);
g#4 = RotateZ(g#3, Angle, g#3[0]);
Output(g#2, g#3, g#4);

gNew = CutAndTransform(Red, g#2, g#3[0], g#3[1], Angle);
Output(gNew);
```



Picture 26

Direction

Calculates direction of the line defined by two points. The result is a point that defines normalized direction vector.

Call format:

```
<id> = Direction(<p1>, <p2>);
```

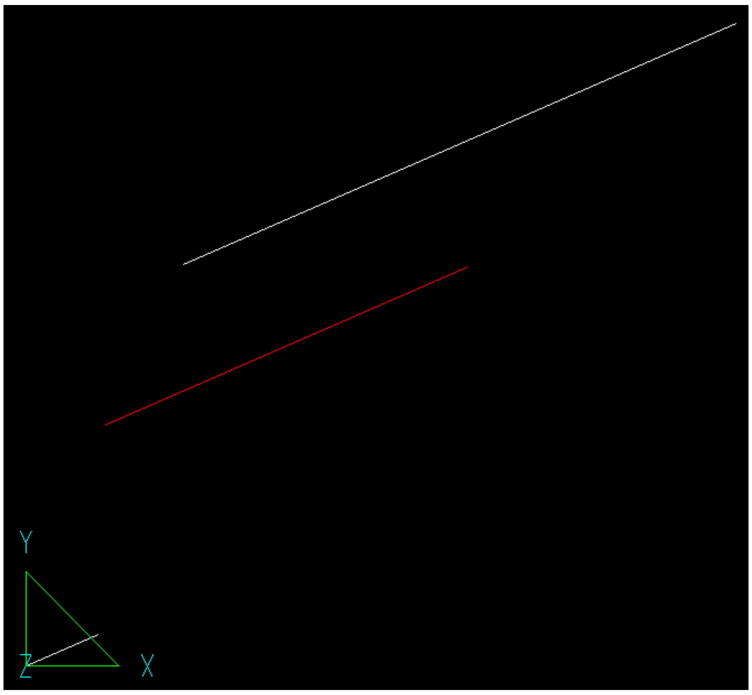
Arguments:

p1, p2 - points that define the line.

Example of using the function:

```
//***Property Sets***
Red = PropertySet("Layer" = "Red", "PenColor" = 0x000000FF);
//***End***

p1 = Point(2, 5);
p2 = Point(9, 8);
g = Polyline(p1, p2);
p3 = Point(1, 3);
length = 5;
Normal = Polyline(Point(0, 0), Direction(p1, p2));
poly = Polyline(Red, p3, p3 + length * Direction(p1, p2));
Output(g, Normal, poly);
```



Picture 27

DirectionToLeft

Calculates perpendicular vector to the line defined by two points. The result is a point, that defines normalized direction vector to the left side of the line.

Call format:

```
<id> = DirectionToLeft(<p1>, <p2>);
```

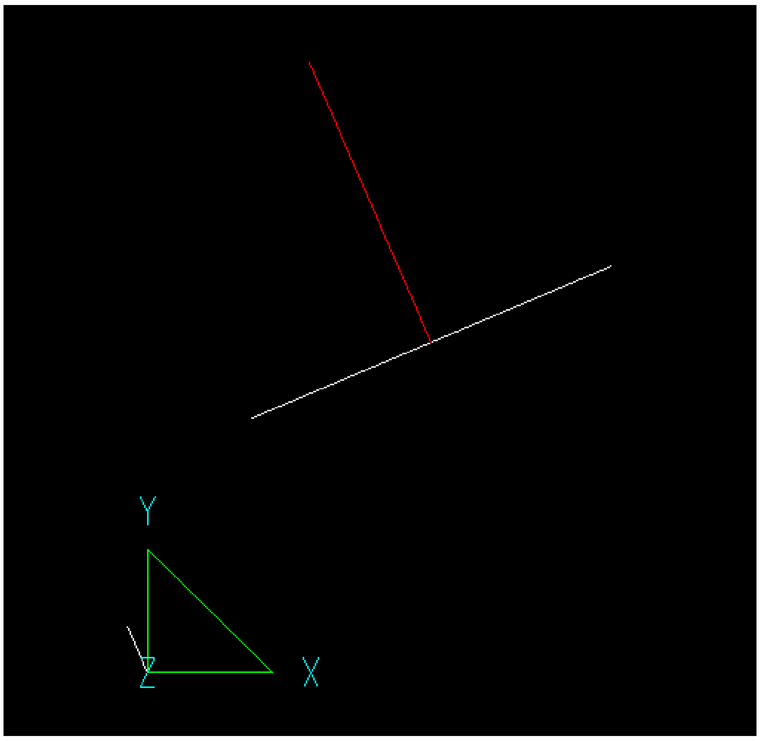
Arguments:

p1, p2 - points that define the line.

Example of using the function:

```
//***Property Sets***
Red = PropertySet("Layer" = "Red", "PenColor" = 0x000000FF);
//***End***
length = 6;
p1 = Point(2, 5);
p2 = Point(9, 8);
g = Polyline(p1, p2);
Normal = Polyline(Point(0, 0), DirectionToLeft(p1, p2));
```

```
pB = Polyline(Red,1/2 * (p1 + p2), 1/2 * (p1 + p2) + length * DirectionToLeft(p1, p2));
Output(g, Normal, pB);
```



Picture 28

Dist

Calculates the distance between two points.

Call format:

```
<id> = Dist(<Point1>, <Point2>);
```

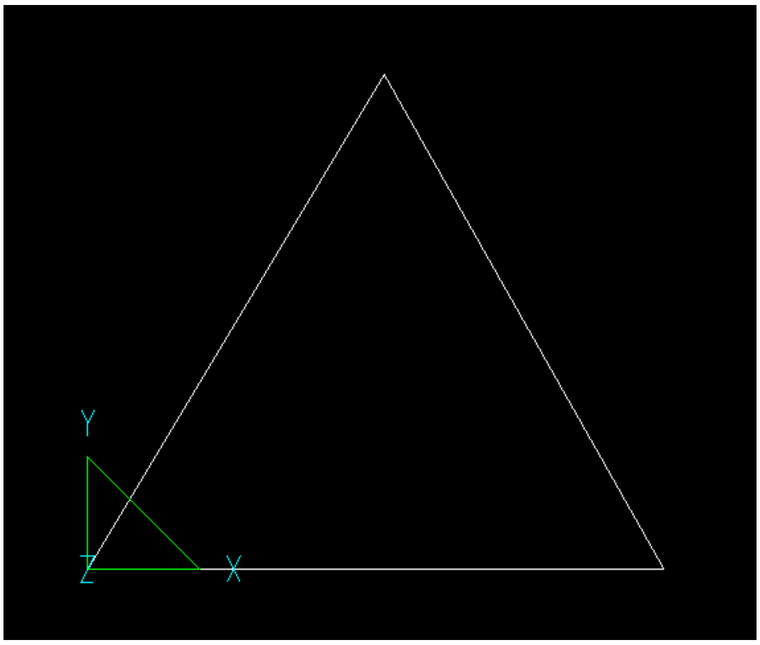
Arguments:

Point1, Point2 - list of point identifiers.

Example of using the function:

```
p#1 = Point(0, 0);
p#2 = Point(3, 5, 0);
p#3 = Point(Dist(p#1, p#2), 0);

Output(Polyline(p#1, p#2, p#3, p#1));
```



Picture 29

ExplodeToPolyline

Transforms 2D objects (circles, arcs, ellipses and polylines with bulge segments) to polyline.

Call format:

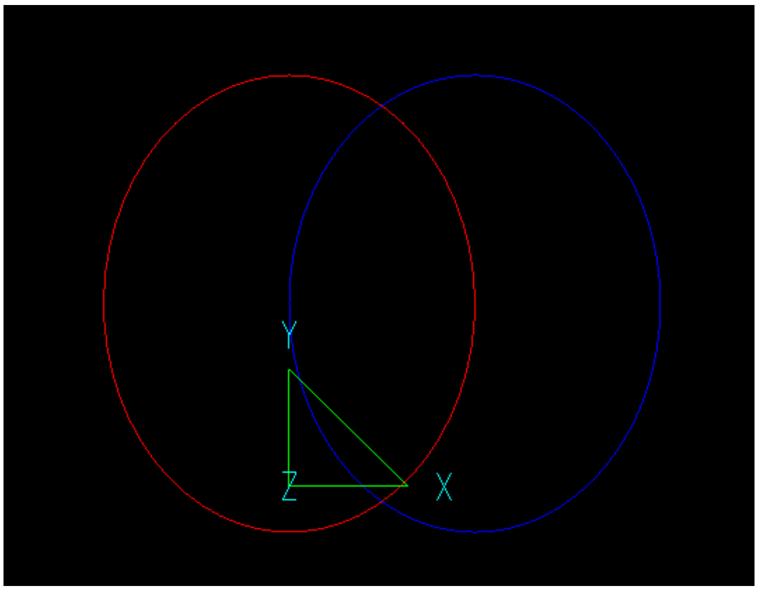
```
<id> = ExplodeToPolyline([<PS>, ]<2D_Object>)
```

Arguments:

PS - Property Style (optional parameter).
2D_Object - 2D object to be transformed.

Example of using the function:

```
//***Property Sets***
Red = PropertySet("Layer" = "Red", "PenColor" = 0x000000FF);
Blue = PropertySet("Layer" = "Blue", "PenColor" = 0x00FF0000);
//***End***
dCenterX = 0; dCenterY = 2;
xSize = 4; ySize = 5;
ellipse = Ellipse(Red, xSize, ySize, dCenterX, dCenterY);
polyEllipse = ExplodeToPolyline(Blue, ellipse);
id = Move(polyEllipse, 2, 0, 0);
Output(ellipse, id);
```



Picture 30

ExtentsX1, ExtentsX2, ExtentsY1, ExtentsY2, ExtentsZ1 and ExtentsZ2

Calculate extents value of an object. ExtentsX1 calculates the minimum extents value of an object along the X-axis. ExtentsX2 calculates the maximum extents value of an object along the X-axis. ExtentsY1 calculates the minimum extents value of an object along the Y-axis. ExtentsY2 calculates the maximum extents value of an object along the Y-axis. ExtentsZ1 calculates the minimum extents value of an object along the Z-axis. ExtentsZ2 calculates the maximum extents value of an object along the Z-axis.

Call format:

```
<id> = ExtentsX1(<gObject>);
<id> = ExtentsX2(<gObject>);
<id> = ExtentsY1(<gObject>);
<id> = ExtentsY2(<gObject>);
<id> = ExtentsZ1(<gObject>);
<id> = ExtentsZ2(<gObject>);
```

Arguments:

gObject - graphical object defined by a name or an expression.

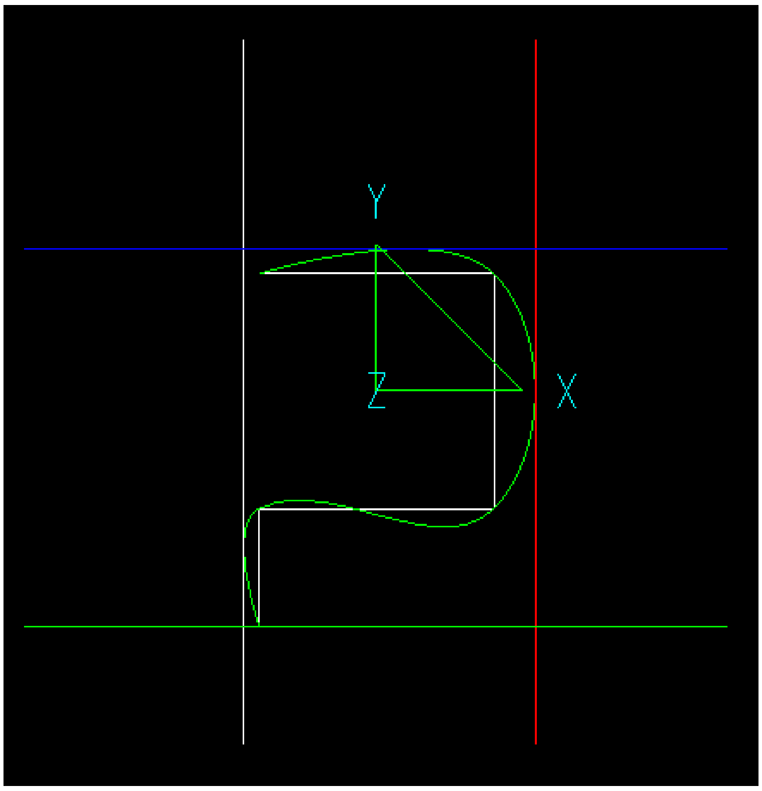
Example of using the function:

```
//***Property Sets***
Red = PropertySet("Layer" = "Red", "PenColor" = 0x000000FF);
Green = PropertySet("Layer" = "Green", "PenColor" = 0x0000FF00);
Blue = PropertySet("Layer" = "Blue", "PenColor" = 0x00FF0000);
//***End***
poly = Polyline(Point(-1, 1), Point(1, 1), Point(1, -1), Point(-1, -1), Point(-1, -2));
b1 = PolyBezier(Green, poly);
Output(poly, b1);
xMin = ExtentsX1(b1);
line1 = Polyline(Point(xMin, -3), Point(xMin, 3));

xMax = ExtentsX2(b1);
line2 = Polyline(Red, Point(xMax, -3), Point(xMax, 3));

yMin = ExtentsY1(b1);
line3 = Polyline(Green, Point(-3, yMin), Point(3, yMin));
```

```
yMax = ExtentsY2(b1);
line4 = Polyline(Blue, Point(-3, yMax), Point(3, yMax));
Output(line1, line2, line3, line4);
```



Picture 31

Flip

Flips an arc.

Call format:

`<id> = Flip([<PS>, ]<gArc>)`

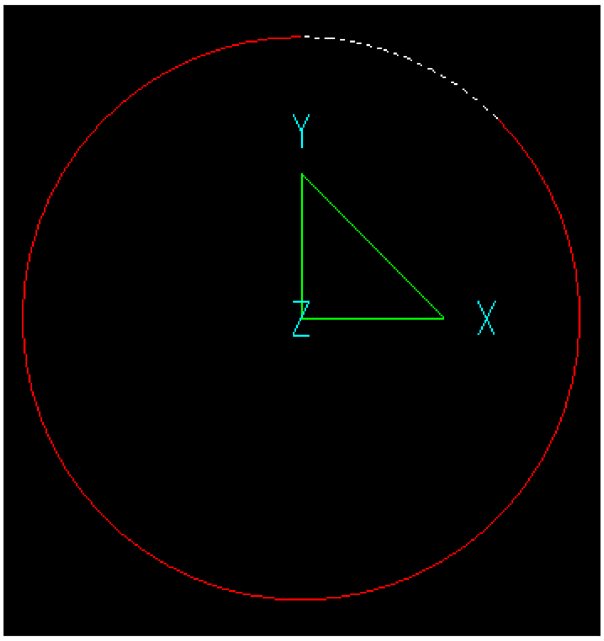
Arguments:

PS - Property Style (optional parameter).

gArc - arc to be flipped.

Example of using the function:

```
//***Property Sets***
Red = PropertySet("Layer" = "Red", "PenColor" = 0x000000FF);
//***End***
arc = Arc (12, 45, 90);
flippedArc = Flip(Red, arc);
Output (arc, flippedArc);
```



Picture 32

Group

Combines several graphical objects into a group. Group allows working with the combined objects as a single object.

Call format:

<id> = Group([<PS>,]<graphic1>, <graphic2>,...);

Arguments:

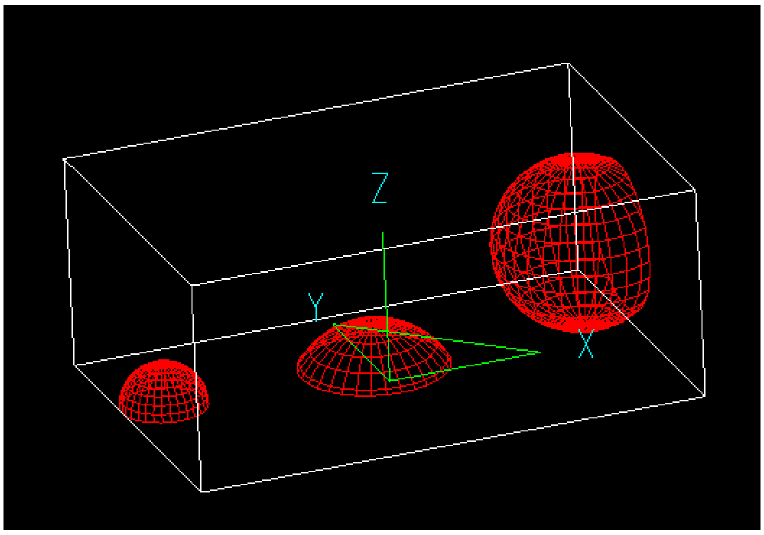
PS - Property Style (optional parameter).
graphic1, graphic2,... - list of graphical objects.

Example of using the function:

```
//***Property Sets***
Red = PropertySet("Layer" = "Red", "PenColor" = 0x000000FF);
//***End***
g1 = Sphere( 2, 5, 1, 2);
g2 = Sphere( 2, 0, 1, -1);
g3 = Sphere( 1, -5, 1, 0);

gr = Group(g1, g2, g3);

b = Thickness(Rectangle(12, 8), 5);
res = BooleanIntersect (Red, b, gr);
Output(res, b);
```



Picture 33

Intersect

Returns a point that is the result of two objects intersection (the closest to pTo)

Call format:

```
<id> = Intersect(<gObject1>, <gObject2>, pTo);
```

Arguments:

gObject1 - graphical object 1.
gObject2 - graphical object 2.
pTo - an auxiliary point that defines the specific point that should be taken as the function return result.

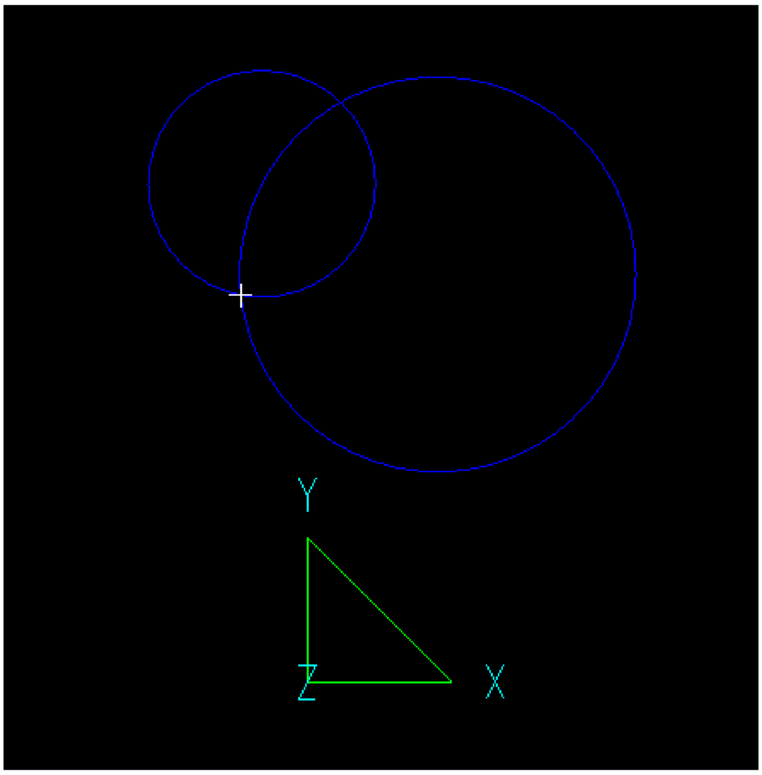
Example of using the function:

```
//***Property Sets***
Blue = PropertySet("Layer" = "Blue", "PenColor" = 0x00FF0000);
//***End***

pCenter = Point(-4, 44, 0); R = 10;
Point1 = Point(27, 28, 0);
Point2 = Point(0, 0);

c01 = Circle(Blue, pCenter, Point1, Point1);
c02 = Circle(Blue, R ,pCenter)
pInt = Intersect(c01, c02, Point2);

Output(c01, c02, pInt);
```



Picture 34

JoinToPolyline

Transforms 2D objects (circles, arcs, ellipses and polylines with bulge segments) to polylines and joins them into a single polyline.

Call format:

<id> = JoinToPolyline([<PS>,]<g1>, <g2>,...)

Arguments:

PS - Property Style (optional parameter).

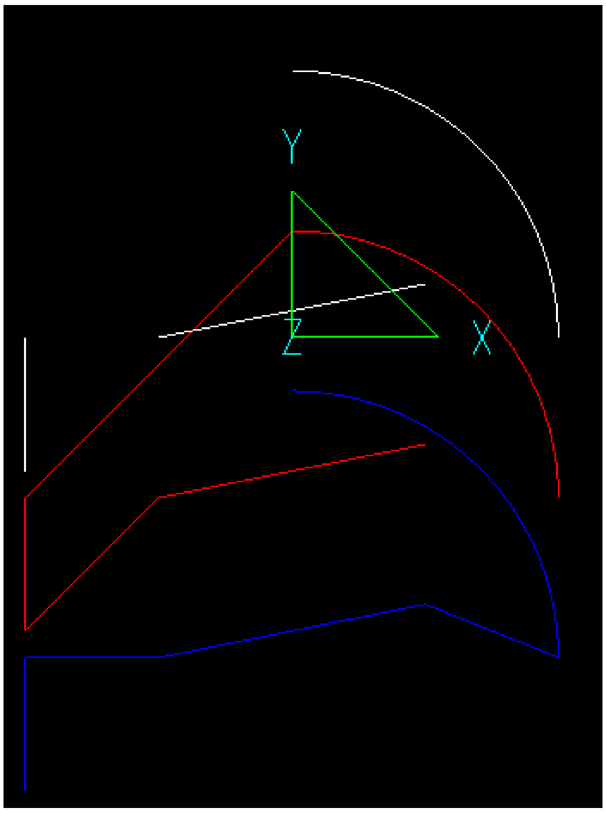
g1, g2,... - 2D objects to be transformed and joined. The order of objects is important.

Example of using the function:

```
//***Property Sets***
Red = PropertySet("Layer" = "Red", "PenColor" = 0x000000FF);
Blue = PropertySet("Layer" = "Blue", "PenColor" = 0x00FF0000);
//***End***

arc = Arc (10, 0, 90);
line1 = Polyline(Point(-10, -5), Point(-10, 0));
line2 = Polyline(Point(-5, 0), Point(5, 2));
JP = Move(JoinToPolyline(Red, arc, line1, line2), 0, -6, 0);

JP2 = Move(JoinToPolyline(Blue, line1, line2, arc), 0, -12, 0);
Output(JP, JP2, arc, line1, line2);
```



Picture 35

LConvexPolyline

Creates "convex" polyline from given polyline, rectifying/deleting hollows on the left side of the polyline.

Call format:

```
<id> = LConvexPolyline([<PS>, ]<gPolyline>[, <gFrame>]);
```

Arguments:

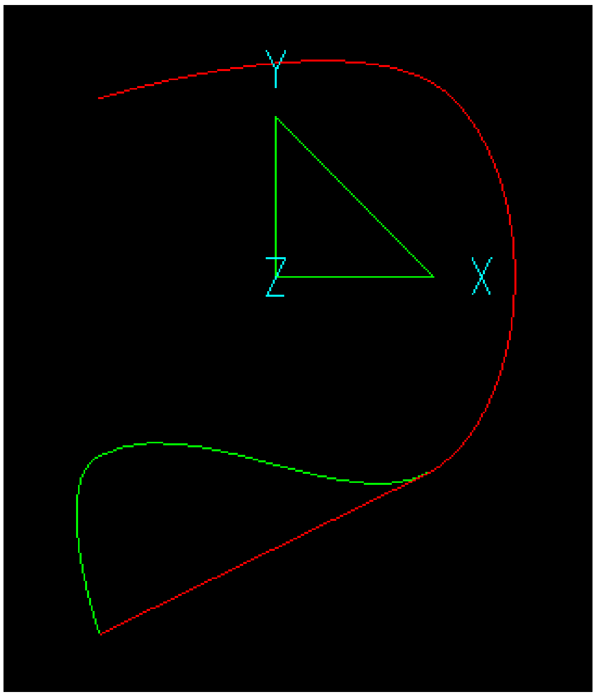
- PS - Property Style (optional parameter).
- gPolyline - source polyline.
- gFrame - polyline with control points (optional parameter).

Example of using the function:

```
//***Property Sets***
Red = PropertySet("Layer" = "Red", "PenColor" = 0x000000FF);
Green = PropertySet("Layer" = "Green", "PenColor" = 0x0000FF00);
//***End***

poly = Polyline(Point(-1, 1), Point(1, 1), Point(1, -1), Point(-1, -1), Point(-1, -2));
b1 = PolyBezier(Green, poly);

bConvex = LConvexPolyline(Red,b1);
Output(b1, bConvex);
```



Picture 36

Length

Calculates length of a 2D object.

Call format:

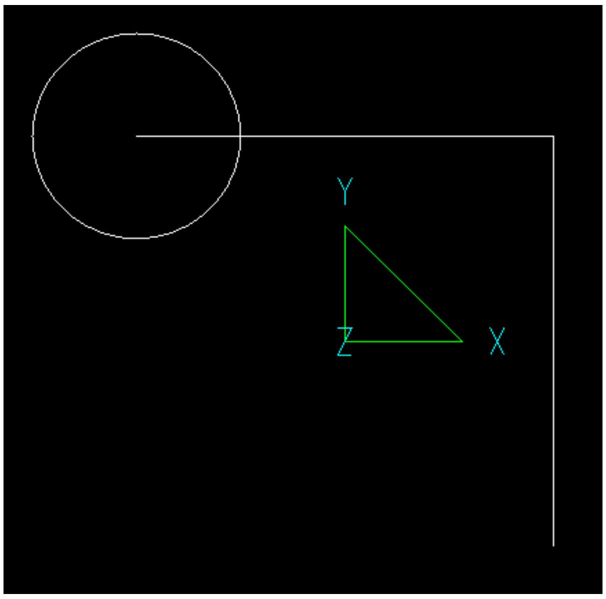
```
<dVal> = Length(<gObj>);
<dVal> = Length(<gObj>, <pTo>);
<dVal> = Length(<gObj>, <pFrom>, <pTo>);
```

Arguments:

gObj - source object.
pFrom - start point on the object for length calculation.
pTo - end point on the object for length calculation.

Example of using the function:

```
poly = Polyline(Point(-1, 1), Point(1, 1), Point(1, -1));
d = Length(poly, Point(-0.5, 1)) ;
g = Circle(d, Point(-1, 1));
Output(b1, poly, g);
```



Picture 37

LineXCircle

Calculates intersection of a line, defined by two points, with a circle, defined by Center point and Radius, that is closest to the specified point.

Call format:

```
<id> = LineXCircle(<p1>, <p2>, <pCenter>, <R>, <pTo>);
```

Arguments:

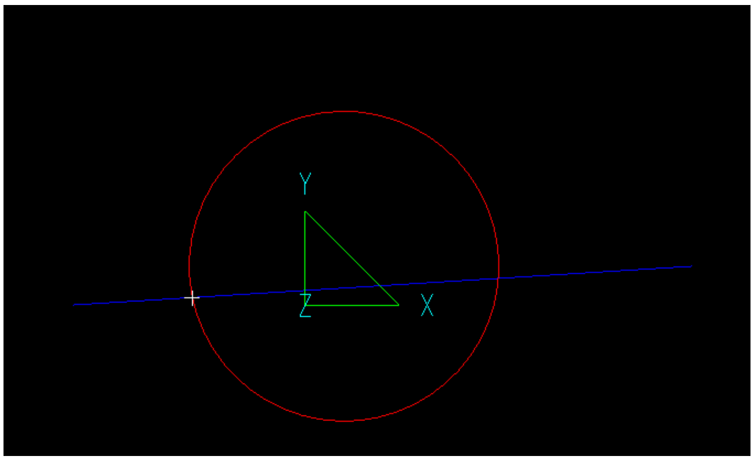
p1, p2 - points that define the line.
pCenter - Circle center point.
R - Radius of the circle.
pTo - point to make choice between variants.

Example of using the function:

```
//***Property Sets***
Red = PropertySet("Layer" = "Red", "PenColor" = 0x000000FF);
Blue = PropertySet("Layer" = "Blue", "PenColor" = 0x00FF0000);
//***End***

l = Polyline(Blue, Point(-6, 0), Point(10, 1));
c2 = Circle(Red, 4, Point(1,1));
Output(l, c2);
pTo = Point(-100, 100);

pResult = LineXCircle(l[0], l[1], Point(1, 1), 4, pTo);
Output(pResult);
```



Picture 38

LineXLine

Calculates intersection of two lines, defined by points.

Call format:

```
<id> = LineXLine(<p1>, <p2>, <p3>, <p4>);
```

Arguments:

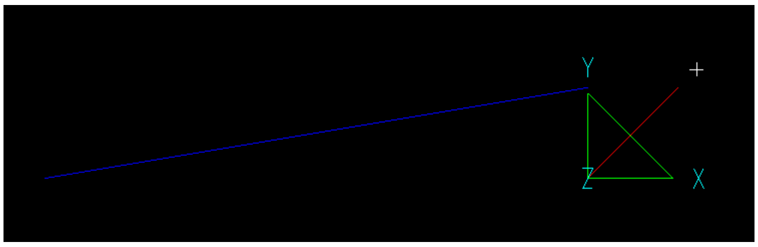
p1, p2 - points that define the first line.
p3, p4 - points that define the second line.

Example of using the function:

```
//***Property Sets***
Red = PropertySet("Layer" = "Red", "PenColor" = 0x000000FF);
Blue = PropertySet("Layer" = "Blue", "PenColor" = 0x00FF0000);
//***End***

l = Polyline(Blue, Point(-6, 0), Point(0, 1));
l2 = Polyline(Red, Point(0, 0), Point(1, 1));
Output(l, l2);
pTo = Point(-100, 100);

pResult = LineXLine(l[0], l[1], l2[0], l2[1], pTo);
Output(pResult);
```



Picture 39

LineXPolyline

Calculates intersection of a line with a polyline.

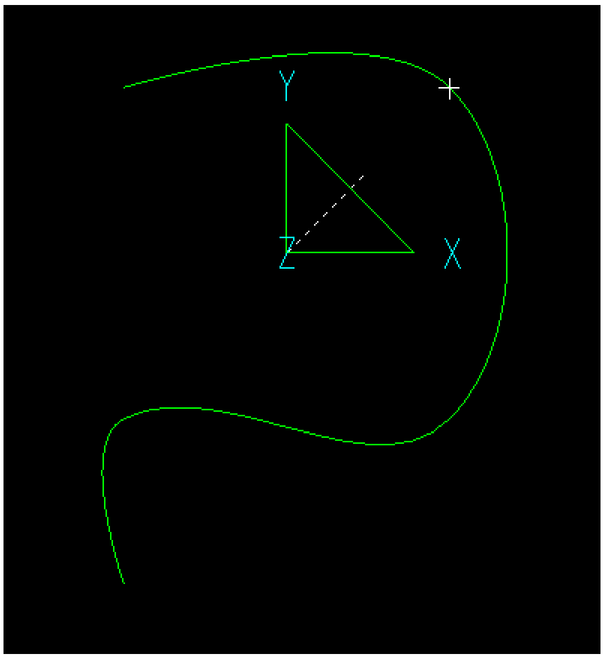
Call format:

```
<id> = LineXPolyline(<p1>, <p2>, <gEntity>);
```

Arguments:
p1, p2 - points that define the line.
gEntity- 2D object (Circle, ellipse, polyline, arc, Bezier curves)

Example of using the function:
```
//***Property Sets***
Green = PropertySet("Layer" = "Green", "PenColor" = 0x0000FF00);
//***End***
polyX = Polyline(Point(0, 0), Point(0.5, 0.5));
poly = Polyline(Point(-1, 1), Point(1, 1), Point(1, -1), Point(-1, -1), Point(-1, -2));
b1 = PolyBezier(Green, poly);

pt = LineXPolyline(polyX[1], polyX[0], b1);
Output(b1, polyX, b1, pt);
```



Picture 40

LinkedPoint

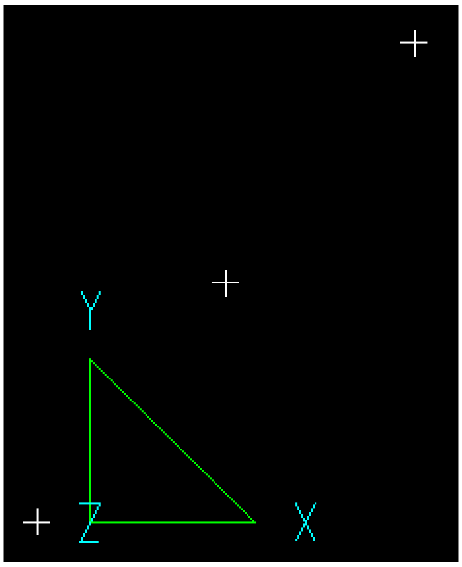
Links a point to 2 base points. Base points define Point Coordinate System (PCS) so the first point has (0,0) coordinates in PCS and the second has (1,0) coordinates in PCS.

Call format:
```
<id> = LinkedPoint(<pS>, <p1>, <p2>);
```

Arguments:

pS - source point.
p1,p2 - base points that define PCS and transformation pS ==> pD.

Example of using the function:
```
p1 = Point(6, 9);
p2 = Point(-1, 0);
pS = Point(0.5, 0);
pD = LinkedPoint(pS, p1, p2);
Output(p1, p2, pD);
```



Picture 41

MaxXPoint and MinXPoint

Calculates a point with the maximum or minimum X coordinate on a 2D entity.

Call format:

```
<id> = MaxXPoint(<gObj>, <pFrom>, <pTo>);
<id> = MinXPoint(<gObj>, <pFrom>, <pTo>);
```

Arguments:

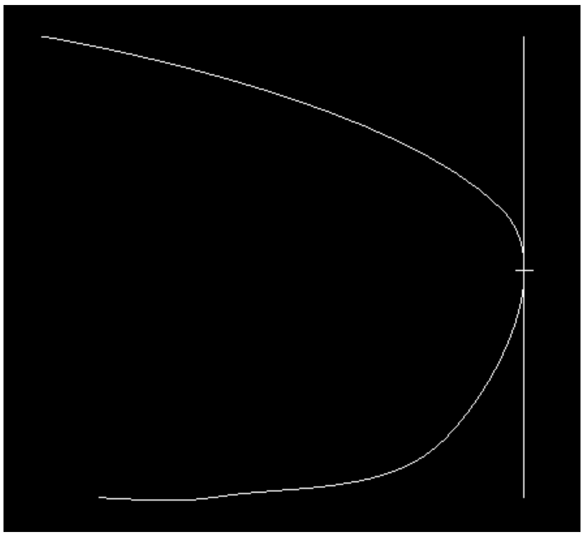
gObj - 2D entity.

pFrom, pTo - points on the entity that define the segment for search.

Example of using the function:

```
p1 = Point(0, -10, 0);
p2 = Point(10, -90, 0);

bezierCrv = PolyBezier(p1, Point(80, -40, 0), Point(70, -80, 0), Point(30, -90, 0), p2);
p = MaxXPoint(bezierCrv, p1,p2 );
Output(bezierCrv, p, Polyline(Point(PointX(p), -10), Offset(0, -80)));
```



Picture 42

MirrorPoint

Calculates a"mirror" of given point relative the line.

Call format:

```
<id> = MirrorPoint(<p1>, <p2>, <pFrom>);
```

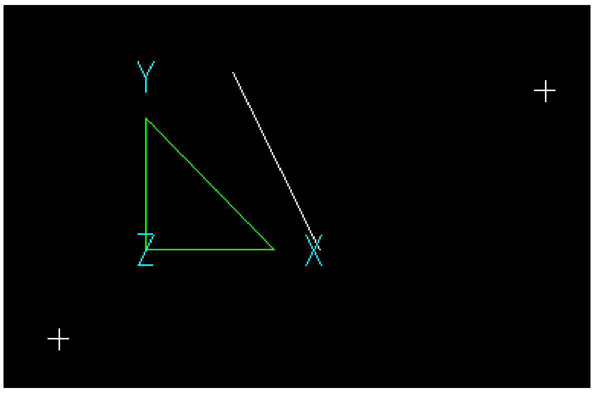
Arguments:

p1, p2 - points that define the line.
pFrom - source point.

Example of using the function:

```
p1 = Point(1, 2);
p2 = Point(2, 0);
pS = Point(-1, -1);
Output(Polyline(p1, p2), pS);

pRes = MirrorPoint(p1, p2, pS);
Output(pRes);
```



Picture 43

NearestPoint

Calculates the point nearest to given point on a 2D entity.

Call format:

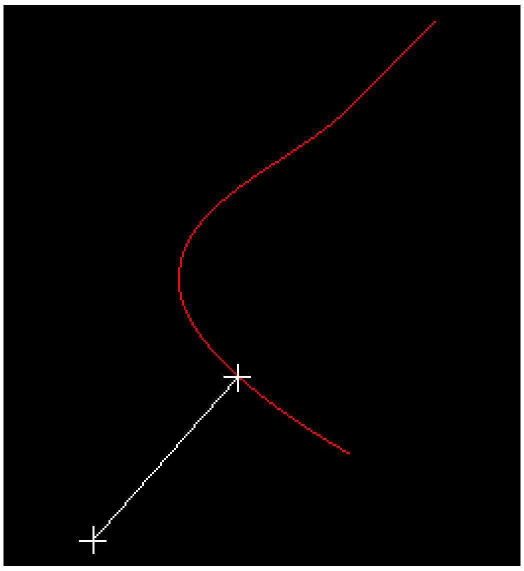
```
<id> = NearestPoint(<gObj>, <pTo>);
```

Arguments:

gObj - 2D entity.
pTo - point to search the nearest to.

Example of using the function:

```
//***Property Sets***
Red = PropertySet("Layer" = "Red", "PenColor" = 0x000000FF);
//***End***
p1 = Point(70, 90, 0);
p2 = Point(60, 80, 0);
p3 = Point(40, 60, 0);
p4 = Point(60, 40, 0);
bezierCrv = PolyBezier(Red, p1, p2, p3, p4);
pTo = Point(30, 30, 0);
pNearest = NearestPoint(bezierCrv, pTo);
poly = Polyline(pTo, pNearest);
Output(poly, bezierCrv, pTo, pNearest);
```



Picture 44

Normal

Finds normal vector to the 2D entity in a given point entity.

Call format:

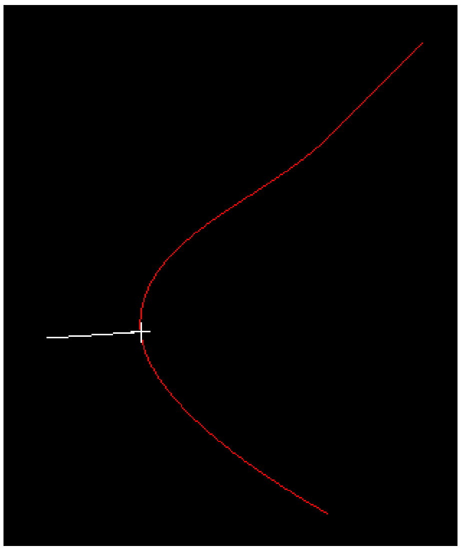
```
<id> = Normal(<gObj>, <pt>);
```

Arguments:

gObj - 2D entity.
pt - given point.

Example of using the function:

```
//***Property Sets***
Red = PropertySet("Layer" = "Red", "PenColor" = 0x000000FF);
//***End***
p1 = Point(7, 9, 0);
p2 = Point(6, 8, 0);
p3 = Point(4, 6, 0);
p4 = Point(6, 4, 0);
bezierCrv = PolyBezier(Red, p1, p2, p3, p4);
pNearest = PointByLength(bezierCrv, p2, 3);
pN = Normal(bezierCrv, pNearest);
poly = Polyline(pNearest, pNearest + pN)
Output(bezierCrv, pNearest, poly);
```



Picture 45

OffsetToPolyline

Creates a polyline offset to the source polyline.

Call format:

```
<id> = OffsetToPolyline([<PS>, ] <gObj>, <dOffset>);
```

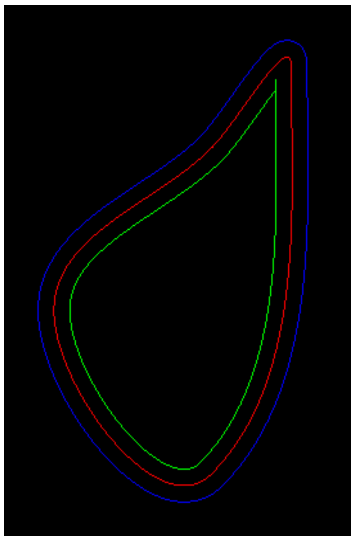
Arguments:

PS - Property Style (optional parameter).
gObj - 2D source object.
dOffset - offset value.

Example of using the function:

```
//***Property Sets***
Red = PropertySet("Layer" = "Red", "PenColor" = 0x000000FF);
Green = PropertySet("Layer" = "Green", "PenColor" = 0x0000FF00);
Blue = PropertySet("Layer" = "Blue", "PenColor" = 0x00FF0000);
//***End***
p1 = Point(70, 90, 0);
p2 = Point(60, 80, 0);
p3 = Point(40, 60, 0);
p4 = Point(60, 40, 0);
bezierCrv = PolyBezier(Red, p1, p2, p3, p4, p1);

gRes = OffsetToPolyline(Blue, bezierCrv, 2);
Output(bezierCrv, gRes);
gRes2 = OffsetToPolyline(Green, bezierCrv,-2);
Output(gRes2);
```



Picture 46

PerpendPoint

Calculates the second point for line segment perpendicular to given line.

Call format:

```
<id> = PerpendPoint(<p1>, <p2>, <pFrom>, <pTo>);
<id> = PerpendPoint(<p1>, <p2>, <pFrom>, <pTo>, <dLength>);
```

Arguments:

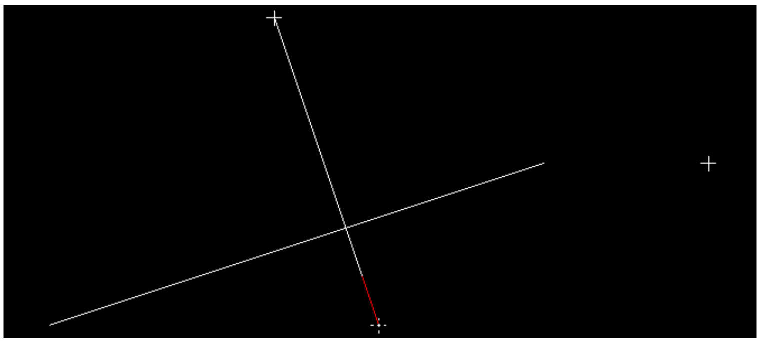
p1, p2 - points that define the line.
pFrom - source point.
pTo - point that defines the second point of the perpendicular.
dLength - length of the perpendicular (optional parameter).

Example of using the function:

```
//***Property Sets***
Red = PropertySet("Layer" = "Red", "PenColor" = 0x000000FF);
//***End***
pFrom = Point(0, 1);
p1 = Point(1, 2); p2 = Point(-2, 1);
pp = Polyline(p1, p2);
Output(pp, pFrom);
dLength = 2;
pTo = Point(2, 2);

Res = PerpendPoint(p1, p2, pFrom, pTo, dLength);
Output(Res, Polyline(Res, pFrom), pTo);

Res2 = PerpendPoint(p1, p2, pFrom, pTo);
Output(Polyline(Red, Res2, pFrom));
```



Picture 47

PointByLength

Finds a point on a 2D entity by distance from given point.

Call format:

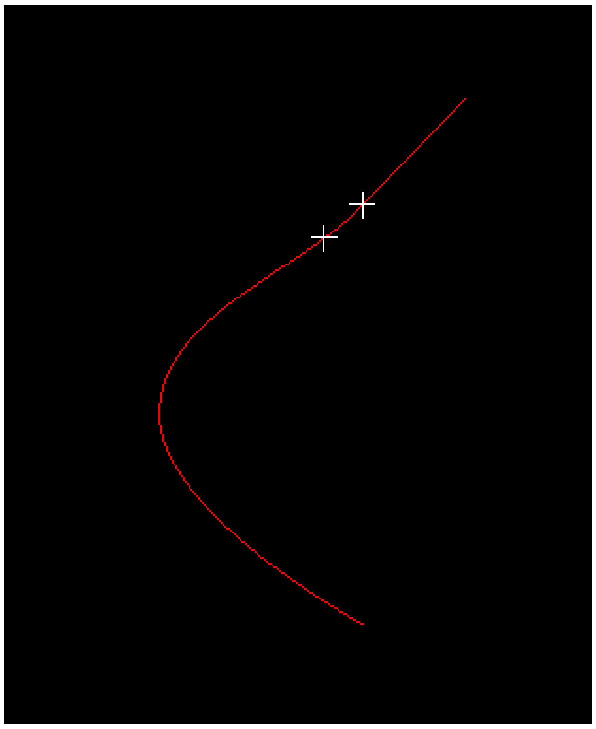
```
<id> = PointByLength(<gObj>, <Length>);
<id> = PointByLength(<gObj>, <pFrom>, <Length>);
```

Arguments:

gObj - 2D entity.
pFrom - start point for the search.
Length – distance from start point.

Example of using the function:

```
//***Property Sets***
Red = PropertySet("Layer" = "Red", "PenColor" = 0x000000FF);
//***End***
p1 = Point(70, 90, 0);
p2 = Point(60, 80, 0);
p3 = Point(40, 60, 0);
p4 = Point(60, 40, 0);
bezierCrv = PolyBezier(Red, p1, p2, p3, p4);
pNearest = PointByLength(bezierCrv, p2, 5);
Output(poly, bezierCrv, p2, pNearest);
```



Picture 48

PointOnPolyline

Finds a point on a 2D entity between two given points.

Call format:

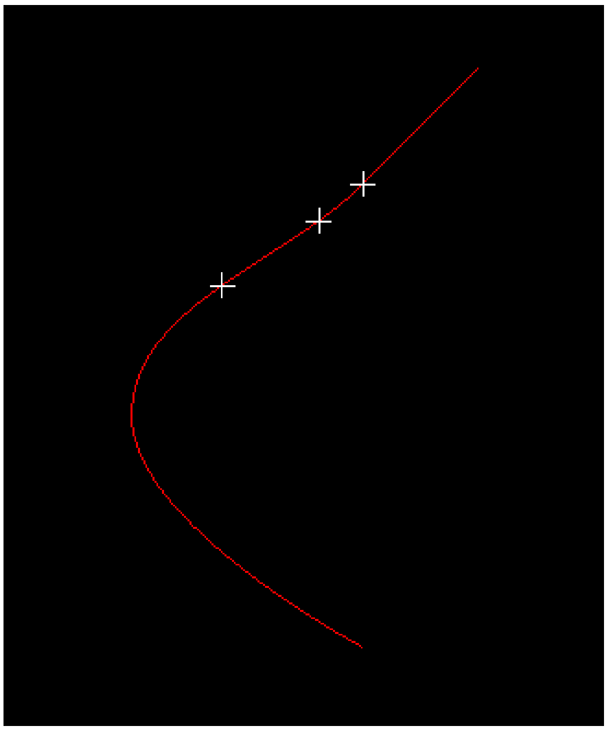
<id> = PointOnPolyline(<gObj>, <p1>, <p2>, <prop>);

Arguments:

gObj - 2D entity.
p1, p2 - start and end points of the segment.
prop – parameter that shows proportion Length(p1, pRes)/Length(p1, p2).

Example of using the function:

```
//***Property Sets***
Red = PropertySet("Layer" = "Red", "PenColor" = 0x000000FF);
//***End***
p1 = Point(70, 90, 0);
p2 = Point(60, 80, 0);
p3 = Point(40, 60, 0);
p4 = Point(60, 40, 0);
bezierCrv = PolyBezier(Red, p1, p2, p3, p4);
pNearest = PointByLength(bezierCrv, p2, 5);
pRes = PointOnPolyline(bezierCrv, p2, pNearest, 0.33);
Output(poly, bezierCrv, p2, pNearest, pRes);
```



ProjectionPoint

Calculates projection of given point onto the line defined by two points.

Call format:

```
<id> = ProjectionPoint(<p1>, <p2>, <pFrom>);
```

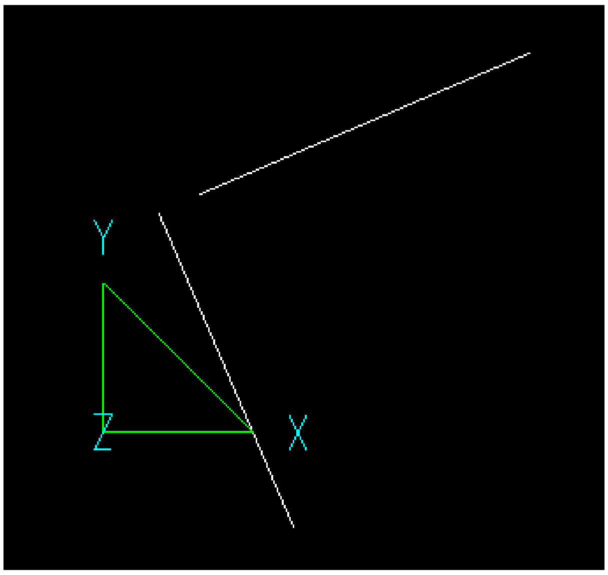
Arguments:

p1, p2 - points that define the line.
pFrom - source point.

Example of using the function:

```
p1 = Point(2, 5);
p2 = Point(9, 8);
g = Polyline(p1, p2);
p3 = Point(4, -2);

Perpendicular = Polyline(p3, ProjectionPoint(p1,  p2, p3));
Output(g, Perpendicular);
```



Picture 49

RConvexPolyline

Creates "convex" polyline from given polyline, rectifying/deleting hollows on the right side of the polyline.

Call format:

<id> = RConvexPolyline([<PS>,]<gPolyline>[, <gFrame>]);

Arguments:

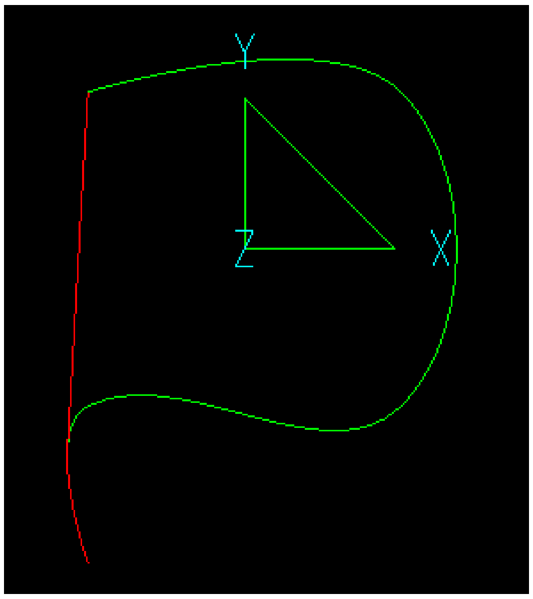
PS - Property Style (optional parameter).
gPolyline - source polyline.
gFrame - polyline with control points (optional parameter).

Example of using the function:

```
//***Property Sets***
Red = PropertySet("Layer" = "Red", "PenColor" = 0x000000FF);
Green = PropertySet("Layer" = "Green", "PenColor" = 0x0000FF00);
//***End***

poly = Polyline(Point(-1, 1), Point(1, 1), Point(1, -1), Point(-1, -1), Point(-1, -2));
b1 = PolyBezier(Green, poly);

bConvex = RConvexPolyline(Red,b1);
Output(b1, bConvex);
```



Picture 50

ReplacePolySegments

Replaces a part of the polyline with another polyline.

Call format:

<id> = ReplacePolySegments([<PS>,]<gPolyline>, <pFrom>, <pTo>, <NewPoly>);

Arguments:

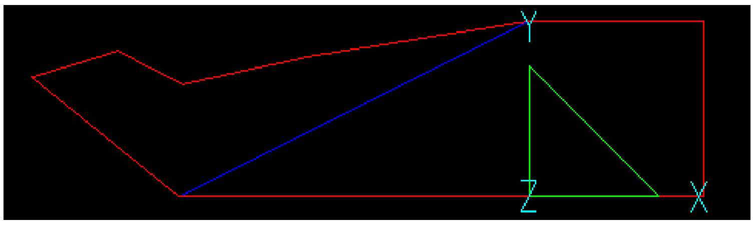
PS - Property Style (optional parameter).
gPolyline - source polyline.
pFrom, pTo - first and second points of the part to be replaced.
NewPoly - polyline for replacement.

Example of using the function:

```
//***Property Sets***
Red = PropertySet("Layer" = "Red", "PenColor" = 0x000000FF);
Blue = PropertySet("Layer" = "Blue", "PenColor" = 0x00FF0000);
//***End***
p1 = Point(1, 1);
p2 = Point(1, 0);
p3 = Point(-2, 0);
p4 = Point(0, 1);
poly = Polyline(Blue, p1, p2, p3, p4, p1);
Output(poly);

p5 = Point(-2.84, 0.68, 0);
p6 = Point(-2.35, 0.83, 0);
p7 = Point(-1.98, 0.64, 0);
p8 = Point(-1.25, 0.8, 0);
p9 = Array(p5, p6, p7, p8);
poly2 = Polyline(p3, p9);
Output(poly2);

gNew = ReplacePolySegments(Red, poly, p3, p4, poly2);
Output(gNew);
```



Picture 51

TangentArc

Creates a tangent arc.

Call format:

```
<id> = TangentArc([<PS>, ]<dRadius>, <g1>, <g2>[, <PointToMakeChoice1>[, <PointToMakeChoice2>]]);
<id> = TangentArc([<PS>, ]<p1>, <p1Dir >, <pCenter>, <dRadCir>);// tangent arc to line1 and circle
<id> = TangentArc([<PS>, ]<p1>, <p1Dir >, <q1>, <q2>[, <p>]);// tangent arc to line1, line2 and point
```

Arguments:

PS - Property Style (optional parameter).
dRadius - arc radius.
dRadCir - circle radius.
pCenter – center point of circle.
g1 - the first circle or arc.
g2 - the second circle.
PointToMakeChoice1, PointToMakeChoice2 - points to choose tangent point.
q1, q2 - start and end line points.
p1 - tangent point.
p1Dir - point that defines arc tangent at point p1.

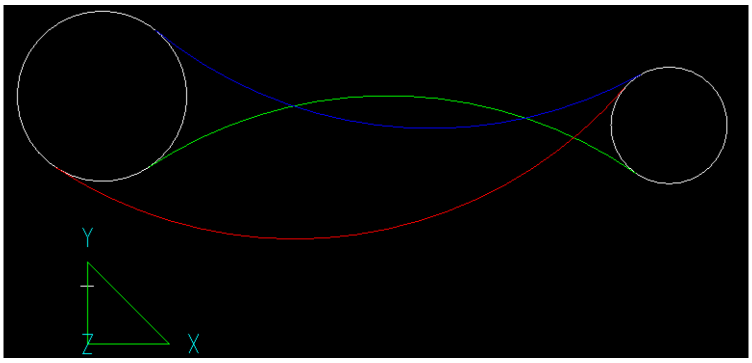
Example of using the function:

```
//***Property Sets***
Red = PropertySet("Layer" = "Red", "PenColor" = 0x000000FF);
Green = PropertySet("Layer" = "Green", "PenColor" = 0x0000FF00);
Blue = PropertySet("Layer" = "Blue", "PenColor" = 0x00FF0000);
//***End***
R = 3;
pCenter = Point(0.1, 1.7);
pRad = Point(0.6, 2);
dRad = Dist(pRad, pCenter);
circle1 = Circle(dRad, pCenter);
Output(circle1);

pCenter2 = Point(4, 1.5);
pRad2 = Point(4, 1.9);
dRad2 = Dist(pRad2, pCenter2);
circle2 = Circle(dRad2, pCenter2);
Output(circle2);

p1 = TangentArc(Blue, R, circle1, circle2);
p2 = TangentArc(Green, R, circle1, circle2, pRad - pCenter, pRad2 - pCenter2);
p3 = TangentArc(Red, R, circle1, circle2, p1[0], p2[1]);

Output(p2, p1, p3);
```



Picture 52

TangentLine

Creates a tangent line.

Call format:

<id> = TangentLine([<PS>,]<gCircle1>, <gCircle2>[, <PointToMakeChoice1>[, <PointToMakeChoice2>]]);
<id> = TangentLine([<PS>,]<gCircle1>, <FromPoint>, <PointToMakeChoice>);

Arguments:

PS - Property Style (optional parameter).
gCircle1, gCircle2 – the first and the second circle.
FromPoint, PointToMakeChoice1, PointToMakeChoice2 - points to choose tangent point.

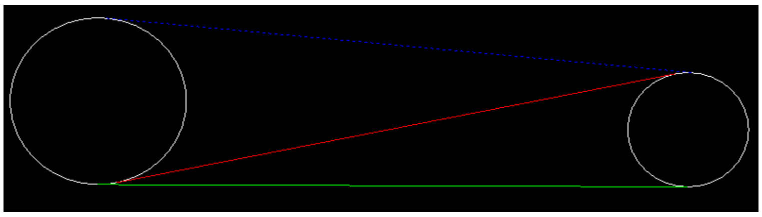
Example of using the function:

```
//***Property Sets***
Red = PropertySet("Layer" = "Red", "PenColor" = 0x000000FF);
Green = PropertySet("Layer" = "Green", "PenColor" = 0x0000FF00);
Blue = PropertySet("Layer" = "Blue", "PenColor" = 0x00FF0000);
//***End***
R = 3;
pCenter = Point(0.1, 1.7);
pRad = Point(0.6, 2);
dRad = Dist(pRad, pCenter);
circle1 = Circle(dRad, pCenter);
Output(circle1);

pCenter2 = Point(4, 1.5);
pRad2 = Point(4, 1.9);
dRad2 = Dist(pRad2, pCenter2);
circle2 = Circle(dRad2, pCenter2);
Output(circle2);

p1 = TangentLine(Blue, circle1, circle2);
p2 = TangentLine(Green, circle1, circle2, pRad - pCenter, pRad2 - pCenter2);
p3 = TangentLine(Red, circle1, circle2, pRad - pCenter, pRad2);

Output(p1 ,p2, p3);
```



Picture 53

TangentPointByDirection

Finds point on a 2D entity by tangent vector.

Call format:

<id> = TangentPointbyDirection([<PS>,]<gObj>, <pDir>[, <pTo>]);

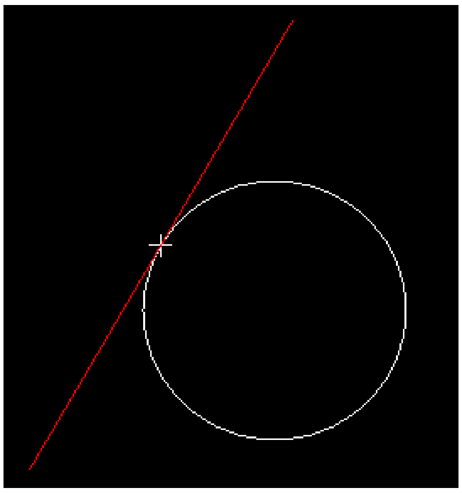
Arguments:

PS - Property Style (optional parameter).
gObj - 2D entity.
pDir - vector(point), that defines tangent direction.
pTo - point that defines, which point should be returned.

Example of using the function:

```
//***Property Sets***
Red = PropertySet("Layer" = "Red", "PenColor" = 0x000000FF);
//***End***
p1 = Point(90,140,0);
p2 = Point(60,190,0);
c = Circle(p1, p2, p2);
Output(c);

A= 60;
q = Point(50,170,0);
q2 = Point(20,120,0);
pDir = Point(cos(A), sin(A));
pTan = TangentPointByDirection(c, pDir, q);
D = Dist(q, q2);
G = Polyline(Red, pTan + D * pDir, pTan - D * pDir);
Output(pTan, G);
```



Picture 54

Creating Dimensions

Dimension

Creates a dimension object.

Call format:

```
<id> = Dimension([<dimStyle>, ]"Angular", <pFL1>, <pFL2>, <pSL1>, <pSL2>,  <pDimLine>);
<id> = Dimension([<dimStyle>, ]"Parallel", <p1>, <p2>,  <pDimLine>);
<id> = Dimension([<dimStyle>, ]"Rotated", <p1>, <p2>,  <pDimLine>, <pDir>);
```

Arguments:

dimStyle - Dimension style as a property set.
"Angular", "Parallel", "Rotated" - dimension type.
p1, p2 – the first and the second point of measured line.
pFL1, pFL2 – the first and the second point of the first angle line.
pSL1, pSL2 – the first and the second point of the second angle line.
pDimLine - a point on the dimension line.
pDir - a point that specifies dimension direction. For OrthoX and OrthoY dimensions, the point coordinates are (1, 0, 0) and (0, 1, 0).

Example of using the function:

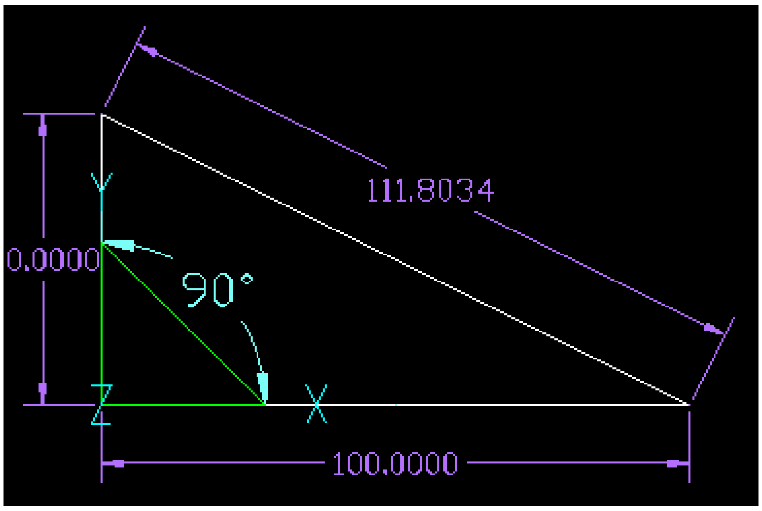
```
//***Property Sets***
dimLinear = PropertySet("PenColor" = 0x00FF70B3, "DimScale" = 20);
dimAngular = PropertySet("PenColor" = 0x00F7FF79, "DimScale" = 30, "DimDec" = 0x00000002);
//***End***

p1 = Point(0, 0, 0);
p2 = Point(100, 0, 0);
p3 = Point(0, 50, 0);
poly = Polyline(p1, p2, p3, p1);
Output(poly);

dimenX = Dimension(dimLinear, "Rotated", p1, p2, Point(p1, PointY(p1) - 10, p1), Point(1, 0, 0));
dimenY = Dimension(dimLinear, "Rotated", p1, p3, Point(PointX(p1) - 10, p1, p1), Point(0, 1, 0));
dimenXY = Dimension(dimLinear, "Parallel", p2, p3, Point((PointX(p2) + PointX(p3)) / 2 + 10, (PointY(p2)
+ PointY(p3)) / 2 + 10, p2));

dimenAng = Dimension(dimAngular, "Angular", p1, Point(PointX(p3) / 2, PointY(p3) / 2, p3), p1,
Point(PointX(p2) / 2, PointY(p2) / 2, p2), Point((PointX(p2) + PointX(p3)) / 4, (PointY(p2) + PointY(p3)) / 4,
p2));

Output(dimenX, dimenY, dimenXY, dimenAng);
```



Picture 55

Creating Image and External Symbol (Not Supported in AutoCAD)

Image

Creates an Image object.

Call format:

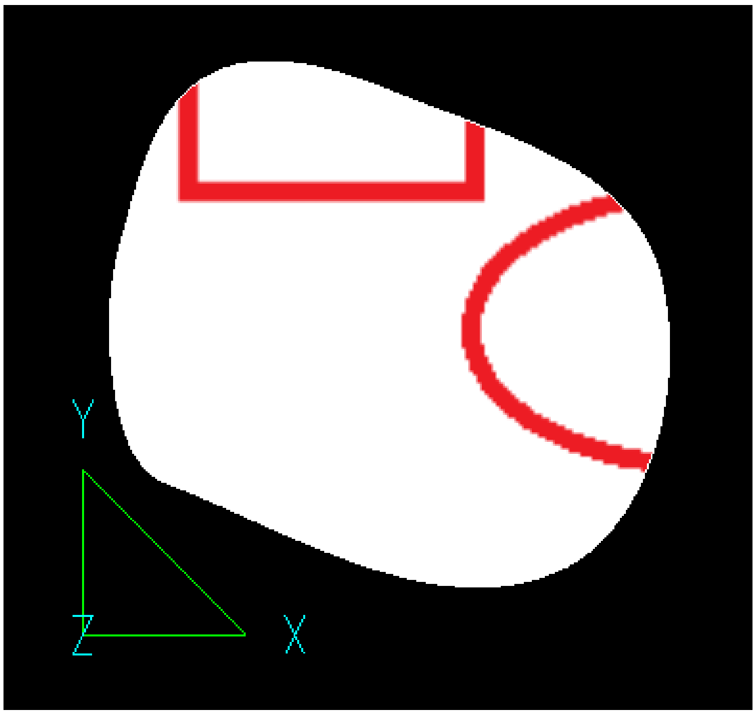
<id> = Image([<PS>,]<gImageDef>, <pPosition>, <dWidth>[, <dHeight>[, <vXAxis>[, <vYAxis>]]]);

Arguments:

PS - Property Style (optional parameter).
gImageDef - Image definition created by ImageDef function.
pPosition - lower-left corner.
dWidth - image width, if it equals 0, the width will be defined by height and aspect ratio.
dHeight - image height (optional parameter) if it is not specified or equals 0, the height will be defined by width and aspect ratio.
vXAxis - image width vector (optional parameter) if it is not defined, it is 0X axis.
vYAxis - image height vector (optional parameter) if it is not defined, it is 0Y axis, or will be defined by image width vector and 0Z axis.

Example of using the function:

```
//image with name 1.png should be in folder C:\temp\
p1 = Point(0, 0);
p2 = Point(10, 0);
q1 = Point(0.56, 1.04);
q2 = Point(0.23, 1.61);
q3 = Point(0.26, 2.67);
q4 = Point(0.86, 3.81);
q5 = Point(2.14, 3.70);
q6 = Point(3.95, 2.55);
q7 = Point(3.29, 0.44);
gClip1 = PolyBezier(q1, q2, q3, q4, q5, q6, q7, q1);
id = ImageDef("C:\temp\1.png");
id2 = Image(id, p1, p2,gClip1);
Output(id2);
```



Picture 56

ImageDef

Creates an Image Definition object.

Call format:

<id> = ImageDef(<sPath>);

Arguments:

sPath – path to image file.

Example of using the function:

See example for Image function.

ImageClip

Clips an Image object.

Call format:

<id> = ImageClip(<gImage>, <p1>, <p2>,…);

Arguments:

gImage - image to be clipped
p1, p2,… - list of 2D points, that bounds the clipping area.

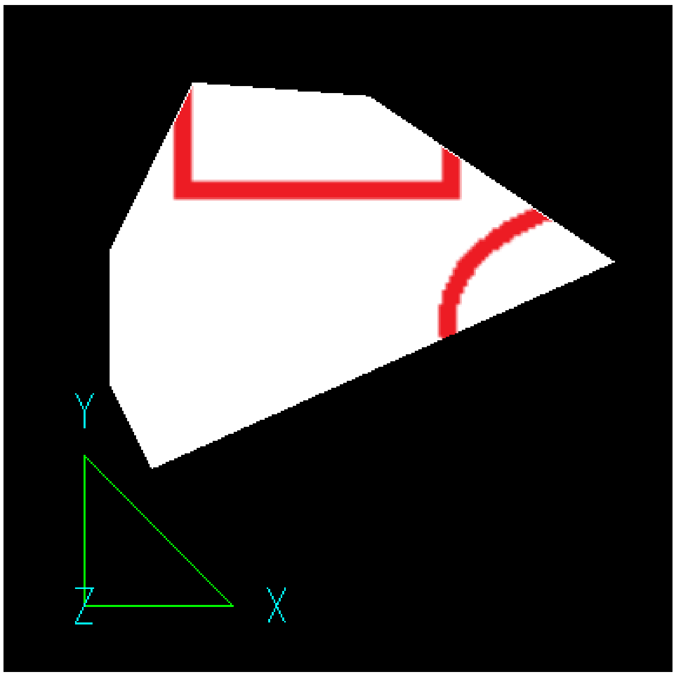
Example of using the function:

```
//image with name 1.png should be in folder C:\temp\
q1 = Point(0.5, 1.0);
q2 = Point(0.2, 1.6);
q3 = Point(0.2, 2.6);
q4 = Point(0.8, 3.8);
q5 = Point(2.1, 3.7);
q6 = Point(3.9, 2.5);

id = ImageDef("C:\temp\1.png");
p1 = Point(0, 0);
```

```
p2 = Point(10, 0);
id2 = Image(id, p1, p2);

id3 = ImageClip(id2, q1, q2, q3, q4, q5, q6);
Output(id3);
```



Picture 57

StaticSymbol

Loads non-parametric symbols from external CAD files.

Call format:

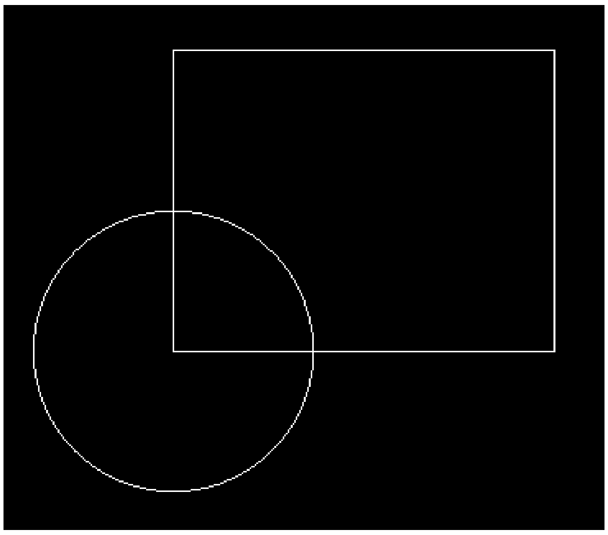
```
<id> = StaticSymbol(<sFileName>, <sBlockName>);
```

Arguments:

sFileName - file name with extension.
sBlockName - name of the block that will be loaded as a symbol.

Example of using the function:

```
Id = StaticSymbol("C:\temp\1.dwg","TestBlock");
Output(Id);
```



Picture 58

Common Functions

Background (Not Implemented)

Sets background color.

Call format:

<id> = Background(<color>);

Arguments:

color – color to be set.

Camera (Not Supported in AutoCAD)

Sets the Camera position. The function allows to create and save a 3D view of objects.

Call format:

<id> = Camera(<pPosition>, <pTarget>, <pUpVector>, <dWidth>, <dHeight>, <bPerspective>);

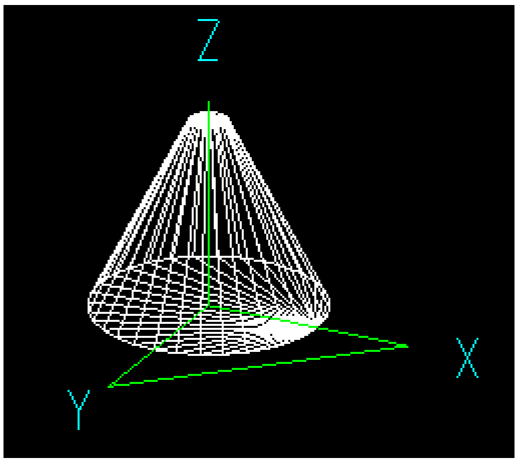
Arguments:

pPosition - point for positioning of the Camera specified with the Point() function.
pTarget - point for Target view specified with the Point() function.
pUpVector - point for defining the Camera orientation in space specified with the Point() function.
dWidth - width of the visible field with the object. Numeric value.
dHeight - height of the visible field with the object. Numeric value.
bPerspective - Boolean flag. If "1" then Camera creates perspective view (Not Implemented yet), if "0"(default value) - then parallel projection is used.

Example of using the function:

```
Conus = Cone (5, 3, 0.5);
Output(Conus);

Camera1 = Camera(Point(-1, -1, 0), Point(0, 1, 1), Point(-0.173275, 0.775195, 0.607493), 20, 35, 0);
Output (Camera1);
```



Picture 59

If

The If function allows various actions to be performed depending upon whether the specified condition is fulfilled or not fulfilled.

Call format:

<id> = If(<condition>, <IfConditionIsTrue>, <IfConditionIsFalse>);

Arguments:

condition – defines the condition that will be checked.
IfConditionIsTrue - defines expression that will be used for calculation of identifier if the condition is true.
IfConditionIsFalse - defines expression that will be used for calculation of identifier if the condition is false.

Example of using the function:

A = If(L >= H, Rectangle(L, H), Rectangle(H, L));

RefPoint (Not Implemented)

Sets the location of the Reference Point for the object. When inserting the object in a drawing, Reference Point allows you to orient the object as needed.

Call format:

<id> = RefPoint(<x>, <y>[, <z>]);

Arguments:

x - coordinates on X-axis.
y - coordinates on Y-axis.
z - coordinates on Z-axes. If z is not specified, it defaults to "0".

RGB

Defines RGB color value.

Call format:

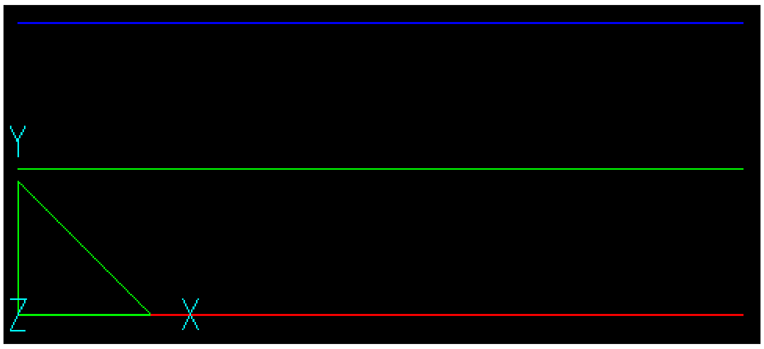
<id> = RGB(<r>, <g>, <b>);

Arguments:

r - red color component (0 - 255)
g - green color component (0 - 255)
b - blue color component (0 - 255)

Example of using the function:

```
//***Property Sets***
Red = PropertySet("Layer" = "Red", "PenColor" = RGB(255, 0, 0));
Green = PropertySet("Layer" = "Green", "PenColor" = RGB(0, 255, 0));
Blue = PropertySet("Layer" = "Blue", "PenColor" = RGB(0, 0, 255));
//***End***
Output(Polyline(Red, Point(0, 0), Point(5, 0)));
Output(Polyline(Green, Point(0, 1), Point(5, 1)));
Output(Polyline(Blue, Point(0, 2), Point(5, 2)));
```



Picture 60

String

Transforms a numerical result returned by a function or an expression into a string.

Call format:

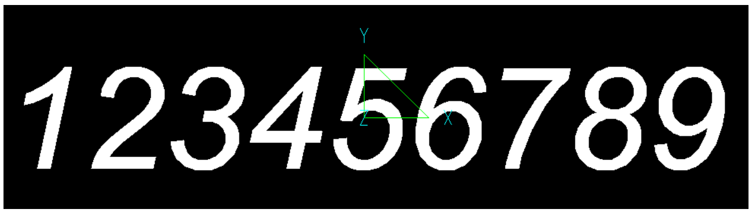
```
<id> = String(<dResult>, <Format>);
```

Arguments:

dResult - graphical object defined by a name or an expression.
Format - output format according to C++ language conventions.

Example of using the function:

```
i = 123456789;
str = String(int(i), "%d");
gsi = Text(str, TextFont(0,0.5,0, "Arial"), TextStyle(CENTER, MIDDLE, ITALIC));
Output(gsi);
```



Picture 61

WorkArea (Not Implemented)

Defines the working area.

Call format:

```
<id> = WorkArea(<pMin>, <pMax>)
```

Arguments:

pMin, pMax – 3D points that define extents of the working area.

Creating 3D Objects Primitives

Cone

Creates a solid Cone or a truncated solid Cone object.

Call format:

```
<id> = Cone([<PS>, ]<dHeight>, <dRadius1>, <dRadius2>);
```

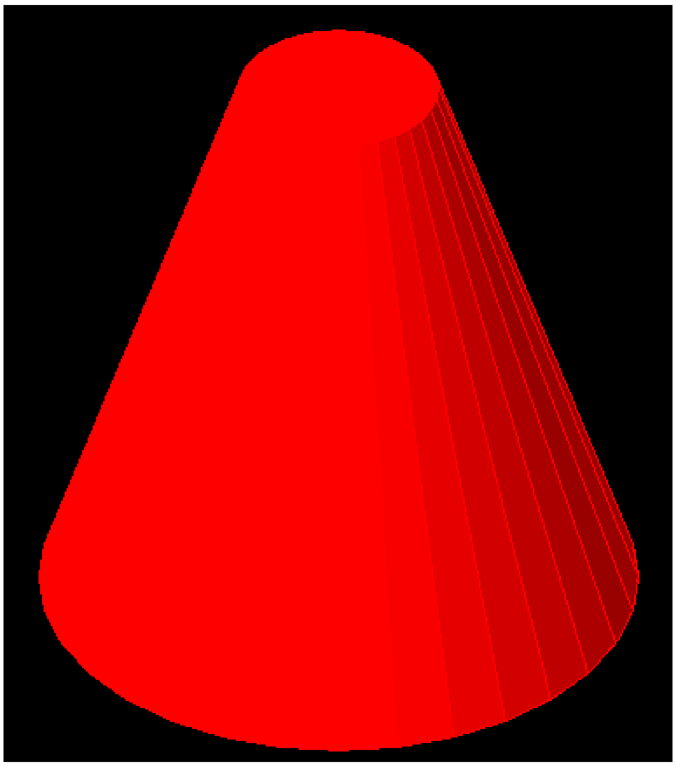
Arguments:

PS - Property Style (optional parameter).
dHeight - cone height value.
dRadius1 - cone base radius value.
dRadius2 - cone top radius value.
Note: The cone base center coordinates are always set to (0, 0, 0).

Example of using the function:

```
//***Property Sets***
Red = PropertySet("Layer" = "Red", "PenColor" = 0x000000FF);
//***End***
dHeight = 6;
dRadius1 = 3;
dRadius2 = dRadius1 - 2;

id = Cone(Red, dHeight, dRadius1, dRadius2);
Output(id);
```



Picture 62

Polyline3D

Creates a 3D Polyline object. In contrast to Polyline function, Polyline3D only includes the line segments and uses Z-coordinate of a point.

Call format:

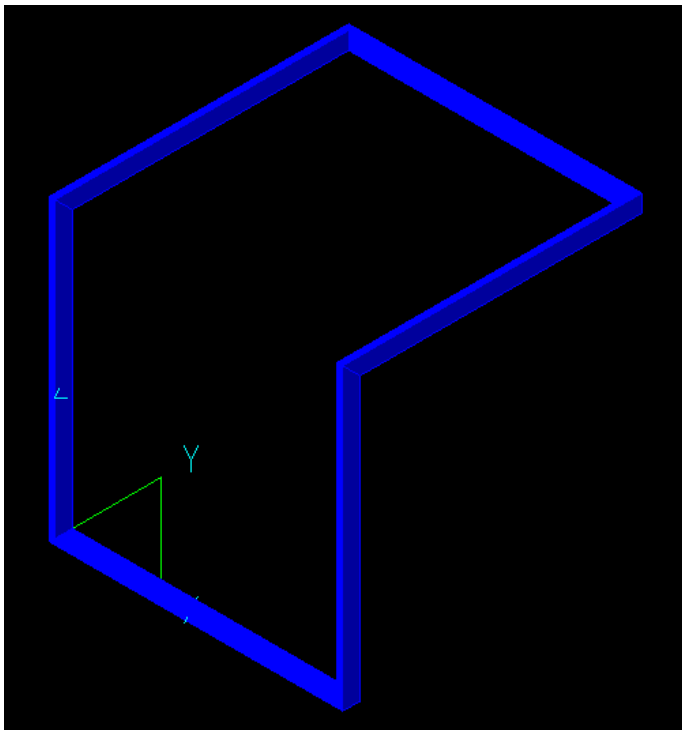
```
<id> = Polyline3D([<PS>, ]<p1>, <p2>,...);
```

Arguments:

PS - Property Style (optional parameter).
p1, p2.. - list of 3D points.

Example of using the function:

```
//***Property Sets***
Blue = PropertySet("Layer" = "Blue", "PenColor" = 0x00FF0000);
//***End***
path = Polyline3D(Blue, Point(0, 0, 0), Point(0, 0, 5), Point(0, 5, 5), Point(5, 5, 5), Point(5, 0, 5), Point(5, 0, 0), Point(0, 0, 0));
profile = Rectangle(0.1, 0.3);
Output(path);
id1 = Sweep(profile, path);
Output(id1);
```



Picture 63

Sphere

Creates a solid Sphere object.

Call format:

```
<id> = Sphere([<PS>, ]<dRadius>, <x>, <y>, <z>);
```

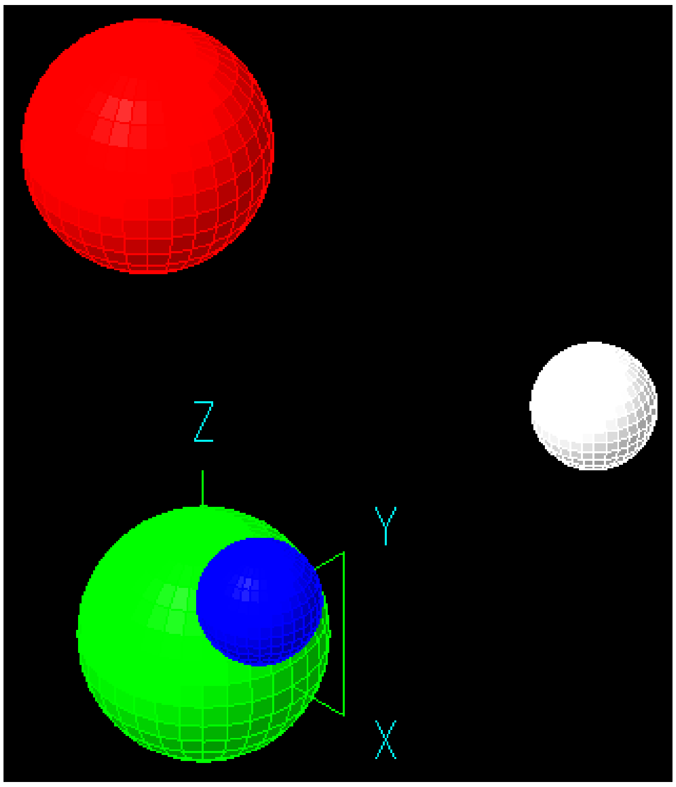
Arguments:

dRadius - the sphere radius value;
x, y, z - coordinates of the sphere center point;
Note: When not specified, the center point coordinates are set to (0, 0, 0).

Example of using the function:

```
//***Property Sets***
Red = PropertySet("Layer" = "Red", "PenColor" = 0x000000FF);
Green = PropertySet("Layer" = "Green", "PenColor" = 0x0000FF00);
Blue = PropertySet("Layer" = "Blue", "PenColor" = 0x00FF0000);
//***End***
```

```
R = 4; x = 10; y = 7.5; z = 10;
s1 = Sphere(R / 2, x, y, z);
s2 = Sphere(Red, R, -x, y, z);
s3 = Sphere(Blue, R / 2, x, -y, z);
s4 = Sphere(Green, R);
Output(s1, s2, s3, s4);
```



Picture 64

Creating 3D Objects from 2D Objects

Loft

Creates a 3D object in the space between 2D cross-section profiles. A profile is defined by a closed 2D polyline.

Call format:

`<id> = Loft([<PS>, ]<ArrayOfProfiles>);`

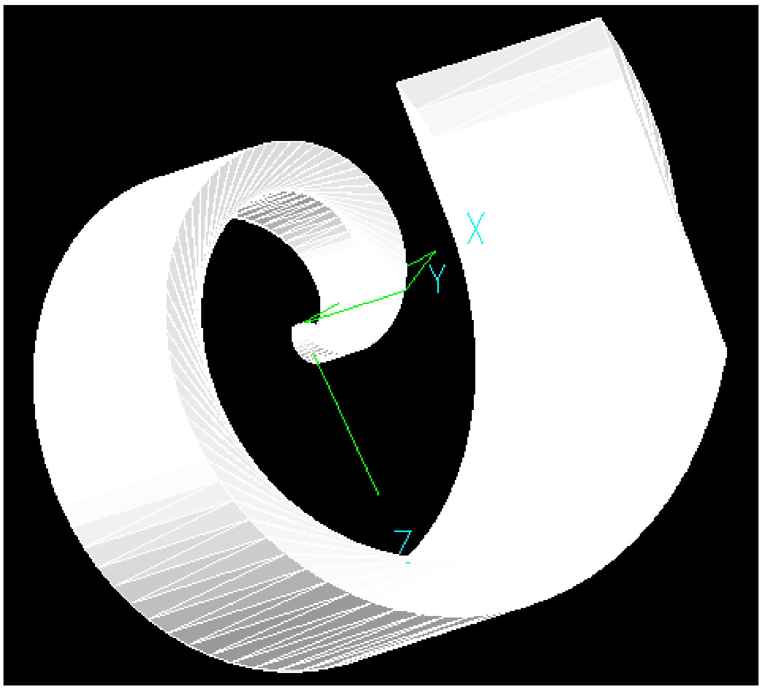
Arguments:

PS - Property Style (optional parameter).
ArrayOfProfiles - array of profiles defining the shape of the resulting object.

Example of using the function:

```
array = MCopy(RotateY(Move(Rectangle(2 + i,4 + i), Point(0, 0, i)), i * 5), i, 0, i < 90);

G_loft = Loft(array);
Output(G_loft);
```



Picture 65

Revolve

Creates a 3D object by revolving a specified profile around an axis. An axis is defined by two points and the rotation angle is specified. Positive angle value means counterclockwise rotation and negative angle value means clockwise rotation. The profile is specified by a closed 2D polyline.

Call format:

```
<id> = Revolve([<PS>, ]<profile>, <p1>, <p2>, <nSegments>, <Angle>);
```

Arguments:

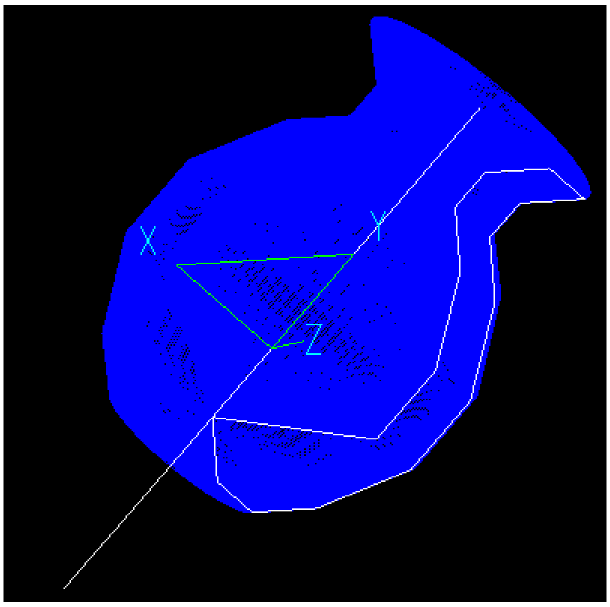
PS - Property Style (optional parameter).
profile - the original profile defined by a 2D polyline.
p1, p2 - the points defining the rotation axis.
nSegments - a number of revolving segments defining the step of the revolution (step = 360/nSegments).
Angle - rotation angle determines how far the profile is revolved around an axis; if not specified, the Angle defaults to 360 degrees.

Example of using the function:

```
//***Property Sets***
Blue = PropertySet("Layer" = "Blue", "PenColor" = 0x00FF0000);
//***End***
poly = Polyline(Point(-20, -30, 0),
Point(-30, -20, 0), Point(-40, 0, 0),
Point(-40, 20, 0), Point(-30, 40, 0),
Point(-20, 50, 0), Point(-20, 60, 0),
Point(-30, 70, 0), Point(-20, 70, 0),
Point(-10, 60, 0), Point(-10, 50, 0),
Point(-20, 40, 0), Point(-30, 20, 0),
Point(-30, 0, 0), Point(0, -20, 0),
Point(-10,  -30, 0), Point(-20, -30, 0));

poly2 = Polyline(Point(0, 70, 0), Point(0, -70, 0));
```

```
id = Revolve(Blue, poly, Point(0, 70,0), Point(0, -70, 0), 300);
Output(id, poly, poly2);
```



Picture 66

Sweep

Creates a 3D object by dragging a specified 2D profile along and perpendicularly to a specified 2D/3D path. The profile is defined by a closed 2D polyline.

Call format:

<id> = Sweep([<PS>,]<profile>, <path>[, <RotationAngle>, <pProfileOriginPoint>]);

Arguments:

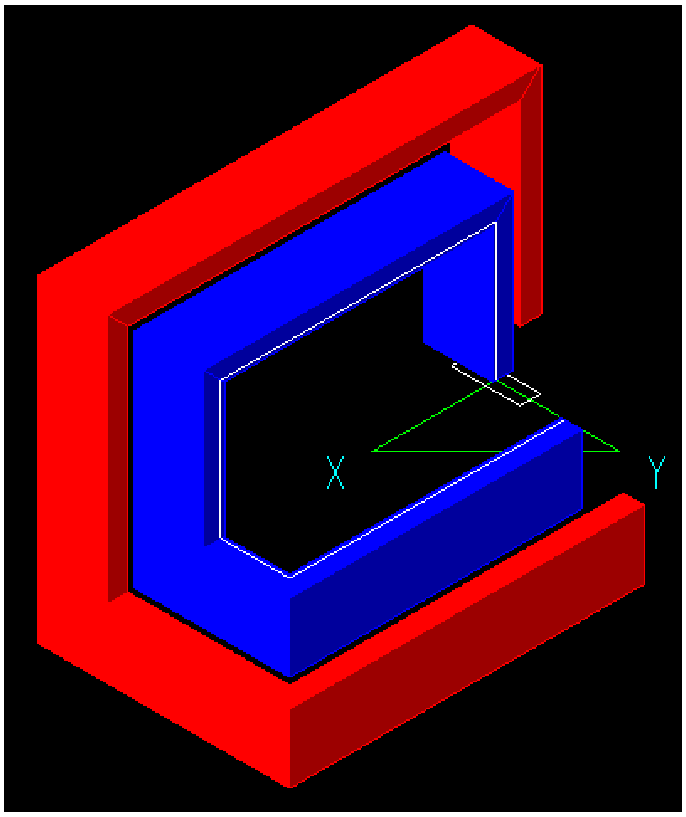
PS - Property Style (optional parameter).
profile - profile defined by a closed 2D polyline.
path - path defined by 2D/3D polyline.
RotationAngle - defines an angle at which the profile should be rotated before dragging.
pProfileOriginPoint - The point on the profile that will be moved in parallel to the path.

Example of using the function:

```
//***Property Sets***
Red = PropertySet("Layer" = "Red", "PenColor" = 0x000000FF);
Green = PropertySet("Layer" = "Green", "PenColor" = 0x0000FF00);
Blue = PropertySet("Layer" = "Blue", "PenColor" = 0x00FF0000);
//***End***
RotationAngle = Parameter("Angle", 0, ANGULAR, Set(0, 30, 45, 90));
pProfileOriginPoint = Point(1, 5, 0);
path = Polyline3D(Point(10, 10, 10), Point(10, 10, 30), Point(50, 10, 30), Point(50, 10, 10), Point(50, 20, 10), Point(10, 20, 10));
profile2 = Rectangle(3,  10);
Output(profile2, path);

s1 = Sweep(Blue, profile2, path, RotationAngle, pProfileOriginPoint);
s2 = Sweep(Red, profile2, path);

Output(s1, s2);
```



Picture 67

Thickness

Creates a 3D object on the basis of a 2D object by adding thickness property. This function allows changing thickness property of a 3D object.

Call format:

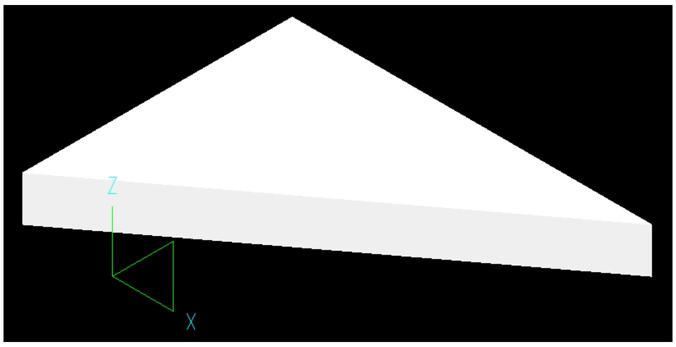
```
<id> = Thickness([<PS>, ]<gObject>, <value>);
```

Arguments:

- PS - Property Style (optional parameter).
- gObject - 2D or 3D object defined by name or expression.
- value - thickness property value.

Example of using the function:

```
g1 = Polyline(Point(-10, 0, 0), Point(-10, 30, 0), Point(30, 30, 0), Point(-10, 0, 0));
id = Thickness(g1, 5);
Output(id);
```



Picture 68

Boolean Operations

Functions of this class are used t9o perform Boolean operations on 3D geometric objects.

BooleanIntersect

Creates a new 3D object from overlapping 3D objects.

Call format:

<id> = BooleanIntersect([<PS>,]<gObject1>, <gObject2>,...);

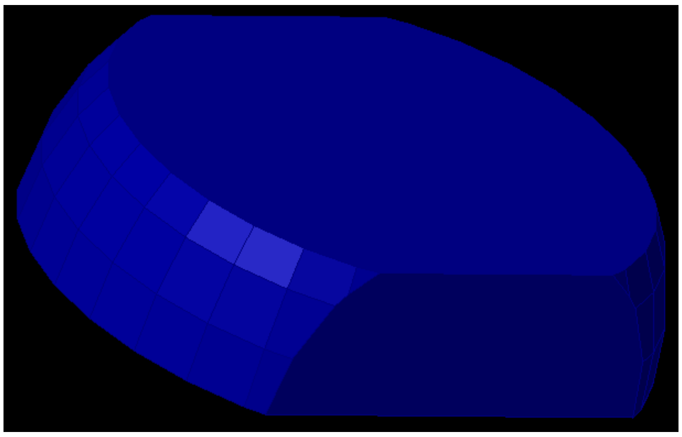
Arguments:

PS - Property Style (optional parameter).
gObject1, gObject2,... - graphical objects, operands of the Boolean operation.

Example of using the function:

```
ob1 = Thickness(Rectangle(20, 15), 5);
ob2 = Sphere(10);

id = BooleanIntersect(("PenColor" = RGB(0, 0, 127)), ob1, ob2);
Output(id);
```



Picture 69

BooleanSubtract

Creates a new 3D object by subtracting one overlapping 3D object from another.

Call format:

<id> = BooleanSubtract([<PS>,]<gObjectMin>, <gObjectSub1>, <gObjectSub2>,...);

Arguments:

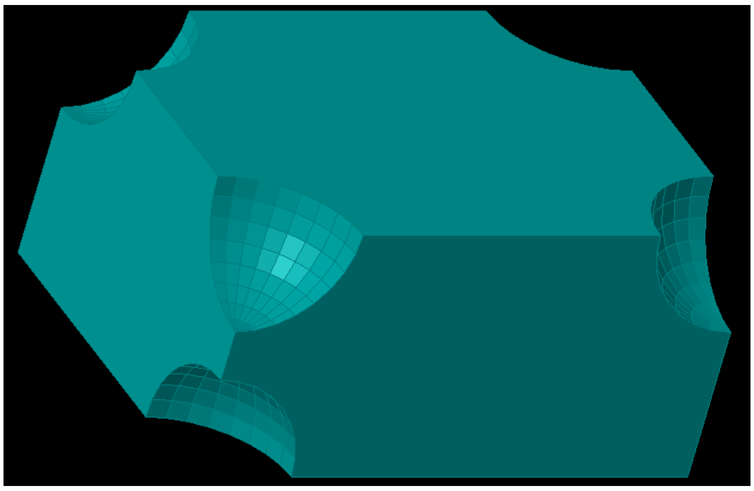
PS - Property Style (optional parameter).
gObjectMin - graphical object that is the minuend of the subtraction operation.
gObjectSub1, gObjectSub2,... - graphical object(s) that is the subtrahend of the subtraction operation.

Example of using the function:

```
ob1 = Thickness(Rectangle(20, 15), 10);
ob2 = Sphere(4, 10, 7.5, 10);
ob3 = Move(ob2, -20, -15, -10);
ob4 = Move(ob2, -20, -15);
ob5 = Move(ob2, -20, 0, 0);
ob6 = Move(ob2, 0, -15, 0);

id = BooleanSubtract(("PenColor" = RGB(0, 127, 127)), ob1, ob2, ob3, ob4, ob5, ob6);
```

```
Output(id);
```



Picture 70

BooleanUnion

Combines two or more 3D objects into a single 3D object.

Call format:

<id> = BooleanUnion([<PS>,]<gObject1>, <gObject2>,...);

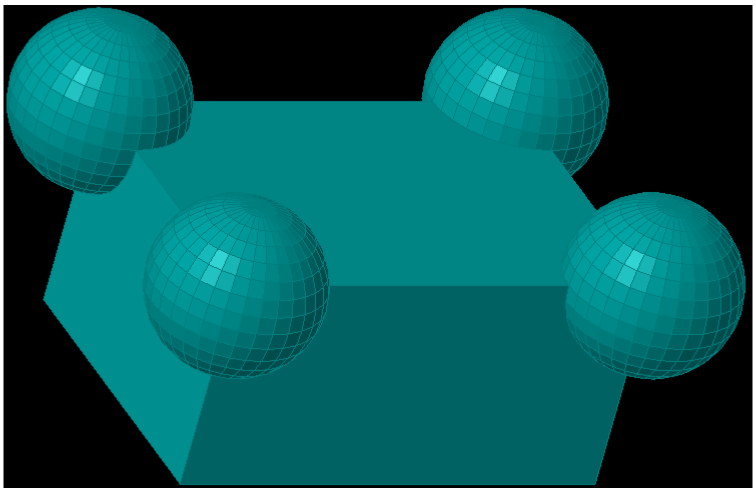
Arguments:

PS – Property Style (optional parameter).
gObject1, gObject2 - graphical objects to be combined.

Example of using the function:

```
ob1 = Thickness(Rectangle(20, 15), 10);
ob2 = Sphere(4, 10, 7.5, 10);
ob4 = Move(ob2, -20, -15);
ob5 = Move(ob2, -20, 0, 0);
ob6 = Move(ob2, 0, -15, 0);

id = BooleanUnion(("PenColor" = RGB(0, 127, 127)), ob1, ob2, ob4, ob5, ob6);
Output(id);
```



Picture 71

Functions to Work with Array

Array

Defines an array of values or an array of geometric objects. Array joins geometric objects or numerical values into an Array object. The objects are numbered starting with 0.

Call format:

```
<id> = Array(<list>);
```

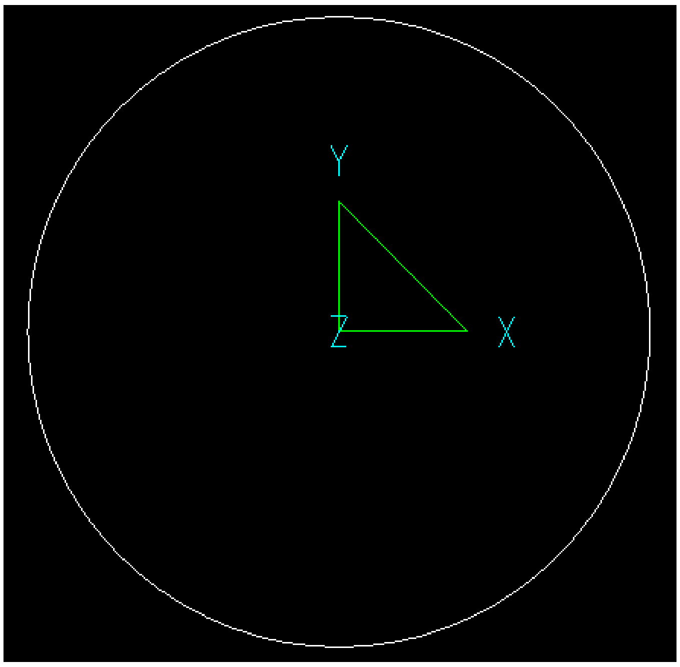
Arguments:

list - list of numerical values or geometric objects.

Example of using the function:

```
Rectangle1 = Rectangle(2, 10);
Circle1 = Circle(1);
Line1 = Polyline(Point(0, 0), Point(15, 14));
n = 1;
a = Array (Rectangle1, Circle1, Line1);

Output(a[n]);
```



Picture 72

ArrayTo3DMesh

Transforms an array of 3D point arrays into a 3D Mesh object.

Call format:

```
<id> = ArrayTo3DMesh([<PS>, ]<DBLArray>);
```

Arguments:

PS - Property Style (optional parameter).

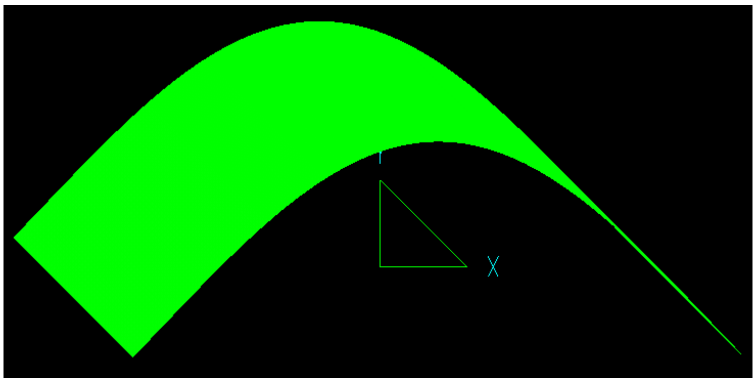
DBLArray - array of 3D Point arrays. Such array can be created, for instance, by MCopy function.

Example of using the function:

```
//***Property Sets***
Green = PropertySet("Layer" = "Green", "PenColor" = 0x0000FF00);
//***End***
```

```
R = 60;
p1 = Point(i + j, R * cos(-i) - j, j);
DBLArray = MCopy(MCopy(p1, i, -100, i < 100),  j, -20, j < 20);

id = ArrayTo3DMesh(Green, DBLArray);
Output(id);
```



Picture 73

ArrayToPolyline

Tranfroms an array of points into a Polyline object.

Call format:

<id> = ArrayToPolyline([<PS>,]<ArrayOfPoints>)

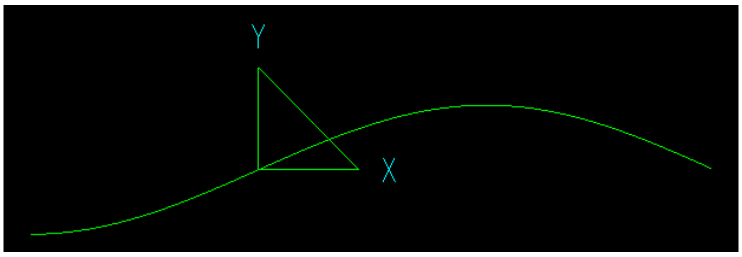
Arguments:

PS - Property Style (optional parameter).
ArrayOfPoints - array of points.

Example of using the function:

```
//***Property Sets***
Green = PropertySet("Layer" = "Green", "PenColor" = 0x0000FF00);
//***End***
R = 25;
p2 = Point(i, R * sin(i));
ArrayOfPoints = MCopy(p2, i, -90, i < 180);

id = ArrayToPolyline(Green, ArrayOfPoints);
Output(id);
```



Picture 74

ArrayToPolyline3D

Tranfroms an array of 3D points into a Polyline3D object.

Call format:

<id> = ArrayToPolyline3D([<PS>,]<ArrayOfPoints>)

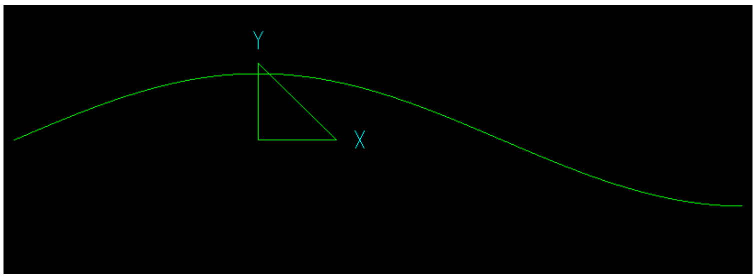
Arguments:

PS - Property Style (optional parameter).
ArrayOfPoints - array of 3D points.

Example of using the function:

```
//***Property Sets***
Green = PropertySet("Layer" = "Green", "PenColor" = 0x0000FF00);
//***End***
R = 25;
p2 = Point(i, R * cos(-i), R * sin(i));
ArrayOfPoints = MCopy(p2, i, -90, i < 180);

id = ArrayToPolyline3D(Green, ArrayOfPoints);
Output(id);
```



Picture 75

MCopy

Creates an Array of objects that are successively calculated by substituting the parameter in the expression that defines the object.

Call format:

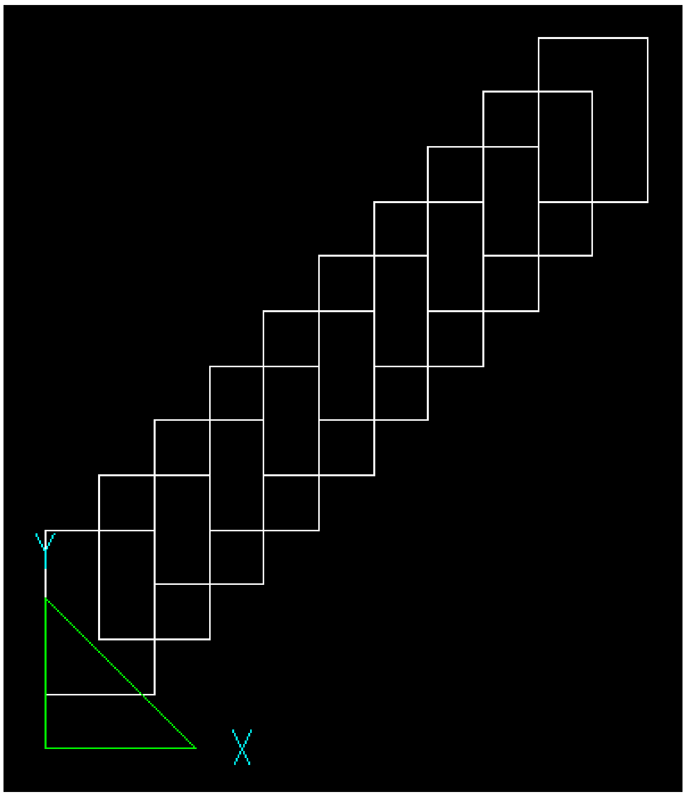
```
<id> = MCopy(<Object>, <LoopVariable>, InitValue, CondExpression);
```

Arguments:

Object - Object that is calculated with the current value of the loop variable; The result of the calculation is added to the array.
LoopVariable - loop variable identifier.
InitValue - initial integer value of the loop variable.
CondExpression - Boolean condition for loop continuation. Object calculation is performed until the condition is true.

Example of using the function:

```
Object = Rectangle(Point(i, 1 + i), Point(i + 2, i + 4));
Array = MCopy(Object, i, 0, i < 10);
Output(Array);
```



Picture 76

SizeOfArray

Returns the array size.

Call format:

<id> = SizeOfArray(<ArrayOfSomething>);

Arguments:

ArrayOfSomething - Array.

Transforming Geometric Objects

MatrixTransform2D

Creates a matrix object by specifying two pairs of vertices. Moves and/or rotates objects by specifying initial and final position for each two points.

Call format:

<id> = MatrixTransform2D(<Point1>, <Point1D>, <Point2>, <Point2D>);
<id> = MatrixTransform2D(<Point1>, <Point2>);

Arguments:

Point1, Point1D - first pair of points (initial and final point position).
Point2, Point2D - second pair of points (initial and final point position).

MatrixTransform2DS

Creates a matrix object by specifying two pairs of vertices. Matrix moves, rotates and scales objects by specifying initial and final position for each two points.

Call format:

<id> = MatrixTransform2DS(<Point1>, <Point1D>, <Point2>, <Point2D>);
<id> = MatrixTransform2DS(<Point1>, <Point2>);

Arguments:

Point1, Point1D - first pair of points (initial and final point position).
Point2, Point2D - second pair of points (initial and final point position).

Move

Copies and moves the graphic objects in the specified direction.

Call format:

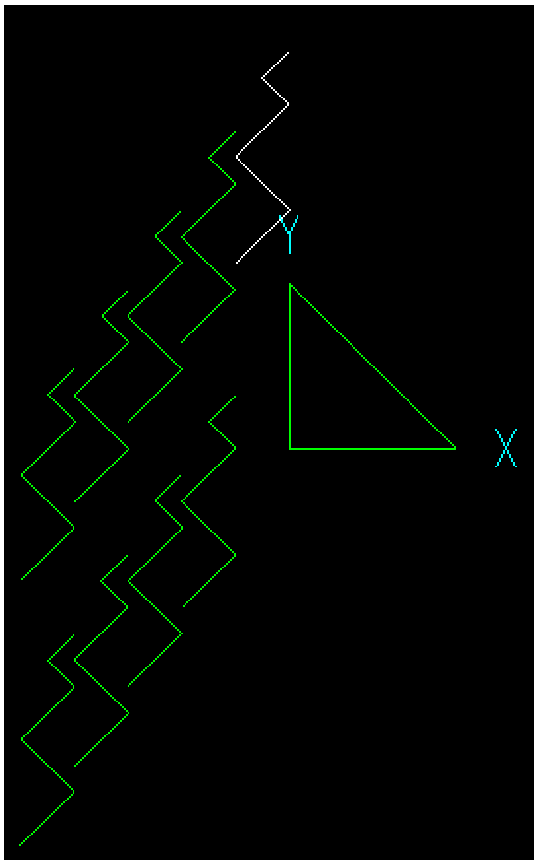
```
<id> = Move([<PS>, ]<gObject>, <dx>, <dy>, <dz>[, <count>]);
<id> = Move([<PS>, ]<gObject>, <pOffset>[,  <count>]);
```

Arguments:

PS - Property Style (optional parameter).
gObject - Graphical object to be moved. Graphical object can be defined by a name or an expression.
dx, dy, dz - coordinates specifying the offset vector.
pOffset - offset vector defined by coordinates.
count - number of objects that will be created by copying and moving the original object. When not specified, the counter value is "1".

Example of using the function:

```
//***Property Sets***
Green = PropertySet("Layer" = "Green", "PenColor" = 0x0000FF00);
//***End***
p1 = Polyline(Point(-2, 7, 0), Point(0, 9, 0), Point(-2, 11, 0), Point(0, 13, 0), Point(-1, 14, 0), Point(0, 15,
0));
id2 = Move(Green, p1, Point(-2, -3, -10), 4);
id3 = Move(id2, 0, -10, 10);
Output(p1, id2, id3);
```



Picture 77

RotateX, RotateY and RotateZ

RotateX rotates graphic objects around an axis in parallel to X-axis. RotateY rotates graphic objects around an axis in parallel to Y-axis. RotateZ rotates graphic objects around an axis in parallel to Z-axis.

Call format:

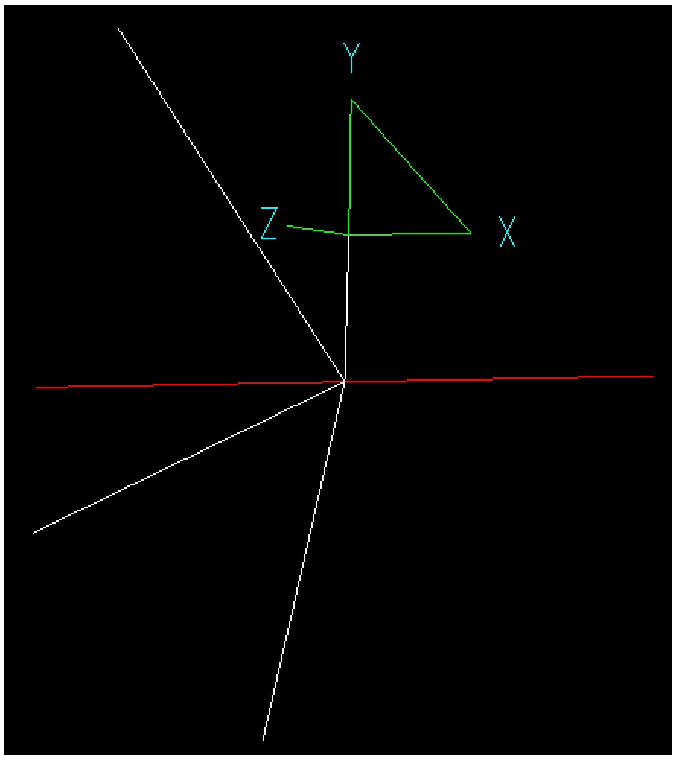
<id> = RotateX([<PS>,]<gObject>, <RotationAngle>, <pAnchor>);
<id> = RotateX([<PS>,]<gObject>, <RotationAngle>, <pAnchor>, <Count>);
<id> = RotateX([<PS>,]<gObject>, <RotationAngle>, <AnchorY>, <AnchorZ>);
<id> = RotateX([<PS>,]<gObject>, <RotationAngle>, <AnchorY>, <AnchorZ>, <Count>);
<id> = RotateY([<PS>,]<gObject>, <RotationAngle>, <pAnchor>);
<id> = RotateY([<PS>,]<gObject>, <RotationAngle>, <pAnchor>, <Count>);
<id> = RotateY([<PS>,]<gObject>, <RotationAngle>, <AnchorY>, <AnchorZ>);
<id> = RotateY([<PS>,]<gObject>, <RotationAngle>, <AnchorY>, <AnchorZ>, <Count>);
<id> = RotateZ([<PS>,]<gObject>, <RotationAngle>, <pAnchor>);
<id> = RotateZ([<PS>,]<gObject>, <RotationAngle>, <pAnchor>, <Count>);
<id> = RotateZ([<PS>,]<gObject>, <RotationAngle>, <AnchorY>, <AnchorZ>);
<id> = RotateZ([<PS>,]<gObject>, <RotationAngle>, <AnchorY>, <AnchorZ>, <Count>);

Arguments:

PS - Property Style (optional parameter).
gObject - original graphical object to be rotated.
RotationAngle - rotation angle value.
pAnchor - a point defining the rotation axis.
AnchorX, AnchorY, AnchorZ - X-coordinate, Y-coordinate and Z- coordinate determinig the fulcrum pin.
Count - number of objects that will be created, so that each next object is added by rotating the previously created one. When not specified, the value is set to "1".

Example of using the function:

```
//***Property Sets***
Red = PropertySet("Layer" = "Red", "PenColor" = 0x000000FF);
//***End***
pAnchor = Point(1, -1, 2);
RotationAngle = 90;
poly = Polyline3D(pAnchor, Point(0, 0, 0));
axis = Polyline3D(Red,
Point(PointX(pAnchor) - 2, PointY(pAnchor), PointZ(pAnchor)),
Point(PointX(pAnchor) + 2, PointY(pAnchor), PointZ(pAnchor)));
id = RotateX(poly, RotationAngle, pAnchor, 3);
Output(poly, id, axis);
```



Picture 78

Scale

Zooms and/or scales the objects linear dimensions.

Call format:

```
<id> = Scale([<PS>, ]<gObject>, <pFixed>, <s>);
<id> = Scale([<PS>, ]<gObject>, <pFixed>, <sX>, <sY>, <sZ>[, <count>]);
```

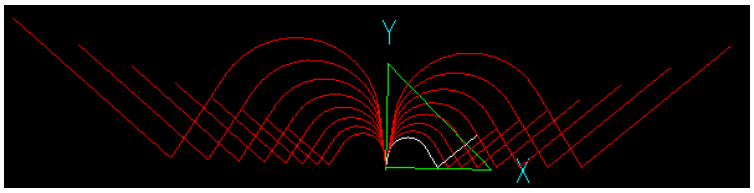
Arguments:

PS - Property Style (optional parameter).
gObject - graphical object to be scaled.
pFixed - center point for scaling.
s - scale factor.
sX, sY, sZ - scale factors for different directions. In the current version these values can differ by sign only. It allows getting mirror transformation.

Example of using the function:

```
//***Property Sets***
Red = PropertySet("Layer" = "Red", "PenColor" = 0x000000FF);
```

```
//***End***
p1 = Point(0, 0);
p2 = Point(0.48, 3.45, 0);
p3 = Point(2.56, 0.06, 0);
p4 = Point(4.48, 1.72, 0);
x = PointX(p4);
y = PointY(p4);
pp = Polyline(p1, p2, Fillet(r), p3, p4);
r=1;
ppp = Scale(Red, pp, Point(0, 0), -1.1, 1.1, -1.1, 15);
Output(ppp, pp);
```



Picture 79

Transform2D

Performs 2D objects transformation by specifying two pairs of vertices. Moves and/or rotates objects by specifying initial and final position for each two points.

Call format:

```
<id> = Transform2D([<PS>, ]<gObject>, <Point1>, <Point1D>, <Point2>, <Point2D>, <Count>);
<id> = Transform2D([<PS>, ]<gObject>, <Point1>, <Point2>, <Count>);
<id> = Transform2D([<PS>, ]<gObject>, <Matrix>, <Count>);
```

Arguments:

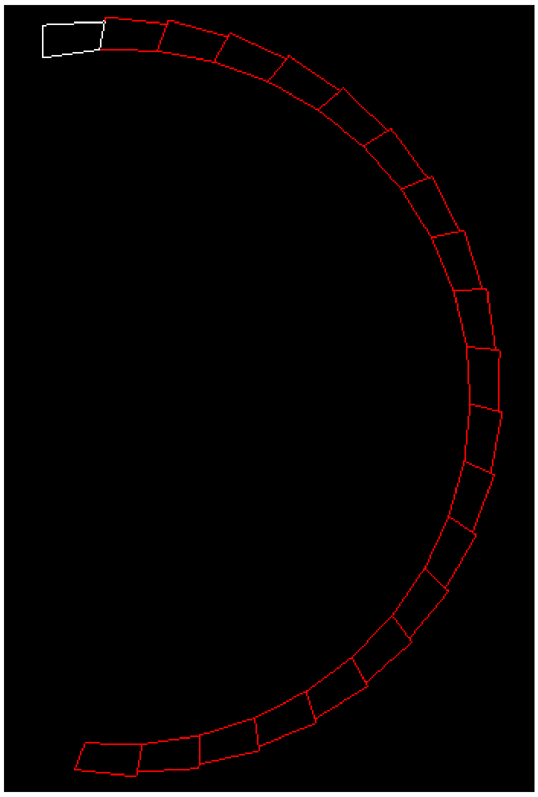
PS - Property Style (optional parameter).
gObject - 2D object to be transformed.
Point1, Point1D - first pair of points (initial and final point position).
Point2, Point2D - second pair of points (initial and final point position).
Matrix - transformation matrix.
Count - number of objects that will be created, so that each next object is added by transforming the previously created one. If the Count value is not specified it is set to "1".

Example of using the function:

```
//***Property Sets***
Red = PropertySet("Layer" = "Red", "PenColor" = 0x000000FF);
//***End***
p1 = Point(-220, -170, 0);
p2 = Point(-100, -150, 0);
p3 = Point(-220, -100, 0);
p4 = Point(-90, -90, 0);

poly = Polyline(p1, p2);
poly1 = Polyline(p3, p4);
arr = Array(p3, p4, p2, p1);
poly2 = Polyline(p1, arr);
Output(poly, poly2);

pp = Transform2D(Red, poly2, p1, p2, p3, p4, 20);
Output(pp);
```



Picture 80

Transform2DS

Performs 2D objects transformation including scaling by specifying two pairs of vertices. Moves, rotates and/or scales objects by specifying initial and final position for each two points.

Call format:

```
<id> = Transform2DS([<PS>, ]<gObject>, <Point1>, <Point1D>, <Point2>, <Point2D>, <Count>);
<id> = Transform2DS([<PS>, ]<gObject>, <Matrix>, <Count>);
```

Arguments:

PS - Property Style (optional parameter).
gObject - 2D object to be transformed.
Point1, Point1D - first pair of points (initial and final point position).
Point2, Point2D - second pair of points (initial and final point position).
Matrix - transformation matrix.
Count - number of objects that will be created, so that each next object is added by transforming the previously created one. If the Count value is not specified it is set to "1".

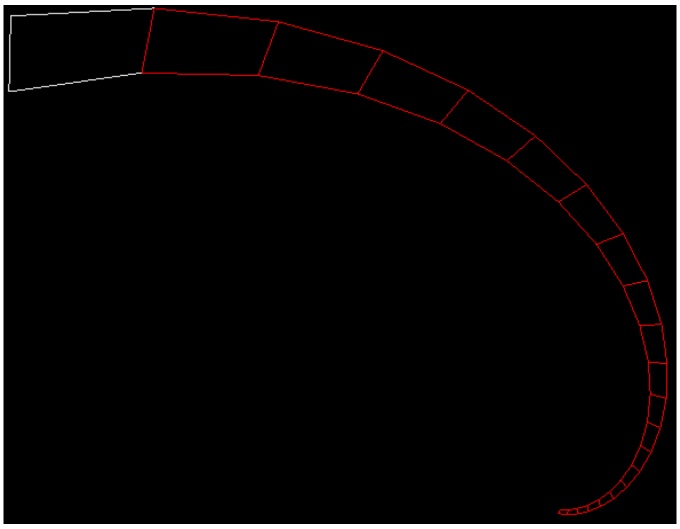
Example of using the function:

```
//***Property Sets***
Red = PropertySet("Layer" = "Red", "PenColor" = 0x000000FF);
//***End***
p1 = Point(-220, -170, 0);
p2 = Point(-100, -150, 0);
p3 = Point(-220, -100, 0);
p4 = Point(-90, -90, 0);

poly = Polyline(p1, p2);
poly1 = Polyline(p3, p4);
arr = Array(p3, p4, p2, p1);
poly2 = Polyline(p1, arr);
```

```
Output(poly, poly2);

pp = Transform2DS(Red, poly2, p1, p2, p3, p4, 20);
Output(pp);
```



Picture 81

Transform3D

Performs 3D transformation by specifying three pairs of vertices. Moves and/or rotates 3D objects by specifying initial and final position for each of three points.

Call format:

<id> = Transform3D([<PS>,]<gObject>, <P1>, <P1D>, <P2>, <P2D>, <P3>, <P3D>[, <count>]);

Arguments:

PS - Property Style (optional parameter).
gObject - graphic to be transformed.
P1, P1D - first pair of points (initial and final positions).
P2, P2D - second pair of points (initial and final positions).
P3, P3D - third pair of points (initial and final positions).
Count - number of objects that will be created, so that each next object is added by transforming the previously created one. If the Count value is not specified it is set to "1".

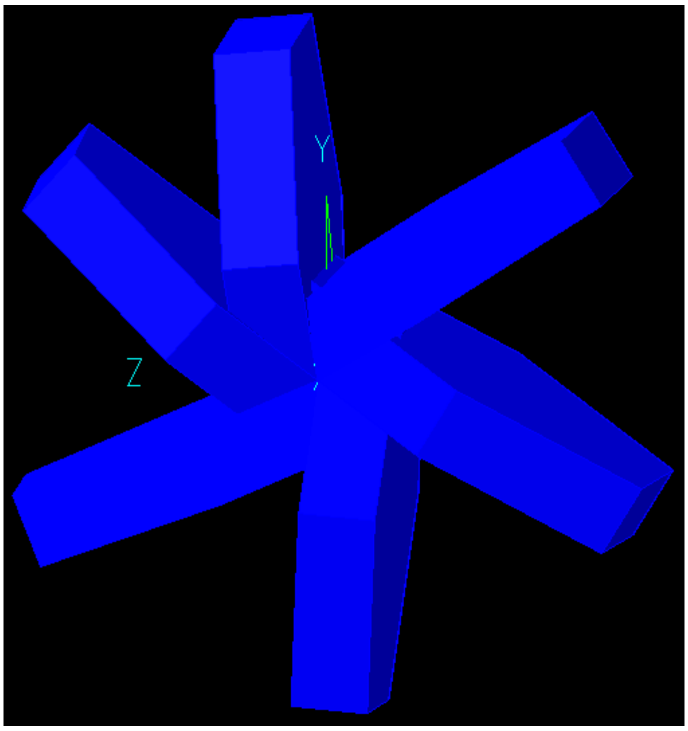
Example of using the function:

```
//***Property Sets***
Blue = PropertySet("Layer" = "Blue", "PenColor" = 0x00FF0000);
//***End***

pC = Point(2.84, 4.44, 0);
pR = Point(10.4, 4.1, 0);
N = 6;
R = Dist(pR, pC);
A0 = AngleXY(pC, pR);
dA = 360 / N;
A = A0 + i * dA;
pi = Point(R * cos(A) + PointX(pC), R * sin(A) + PointY(pC));
pArray = MCopy(pi, i, 0, i <= N);
g = Polyline(Blue, pArray);
g2 = Thickness(Blue, g, 2.5);
```

```
p1 = Point(0, 0);
p2 = Point(1, 0);
p3 = Point(0, 1);
p4 = Point(0, 1, 2);
g3 = Transform3D(g2, p1, p1, p2, p2, p3, p4, 5);

Output(g2,g3);
```



Picture 82

Custom functions

In addition to predefined functions, custom (user) functions can be created. This feature allows avoiding duplication of script text and decreasing error count. PPM language has two formats for defining custom functions.

New Custom Function Format

A function is declared in the following manner:

```
function_name(list_of_arguments) {function_body;}
```

function_name - a sequence of characters used to denote the name of custom function. It is a set of Roman letters and Arabic numerals, which must start with a letter. This name should be unique and may not be the same as predefined functions, directives and keywords.

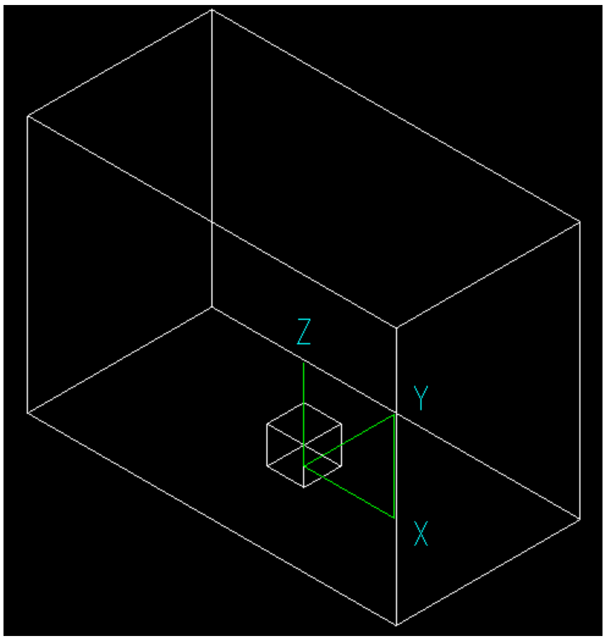
list_of_arguments- a list of function arguments separated with commas.

function_body - sequence of script statements that must contain at least one Output directive with argument.

Example of defining custom function:

```
W = 10;
H = 5;
TH = 7;
Output(Box(W, H, TH), Box(1, 1, 1));
```

```
// Custom function Box
Box(W, H, TH)
{
	rect = Rectangle(W, H);
	box = Thickness(rect, TH);
	Output(box);
}
```



Picture 83

Custom functions of such format can be defined in the main script file (as in the example above) or in a separate script file with **.ppm** extension. The name of the separate file list should be the same as the custom function name. The file with the custom function should be placed in the same folder or in a nested folder with the name **Macro**.

PPM language searches for functions in the following order:

1. In the list of predefined functions.
2. In the text of the calling script.
3. In the folder next to the calling script.
4. In the **Macro** folder nested in the folder with the calling script.

Note: Functions using the new format are useless without a calling script. They won't give any result after being executed by themselves.

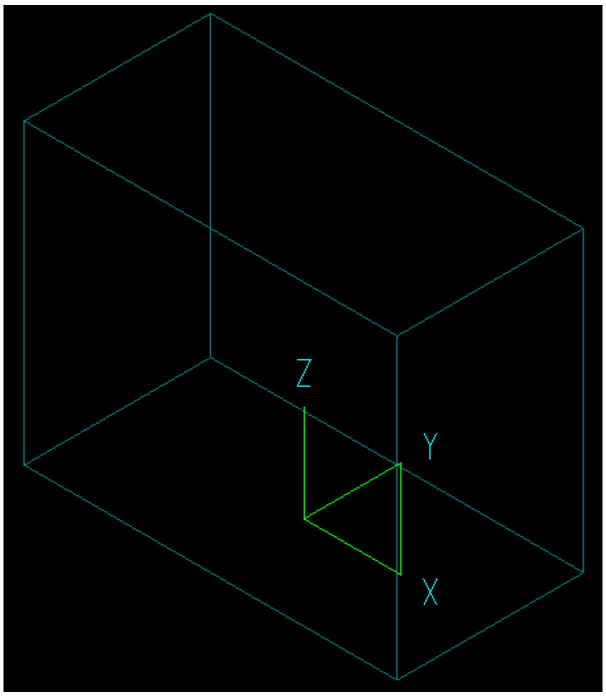
Old Custom Function Format

Custom function of the old format cannot be stored in the same file as the calling script. The name of the file that contains a custom function is the name of the custom function. Custom function file should be placed as described above in the previous topic.

Script of the custom function file should contain an Input directive that defines the list of function arguments and an Output directive that defines the resultant function object.

Example of defining custom function:
File Box.ppm

```
W = Parameter("Width", 10, LINEAR);
H = Parameter("Height", 5, LINEAR);
TH = Parameter("Thickness", 8, LINEAR);
Input(W, H, TH);
rect = Rectangle(W, H);
box = Thickness("PenColor" = RGB(0, 127, 127), rect, TH);
Output(box);
```

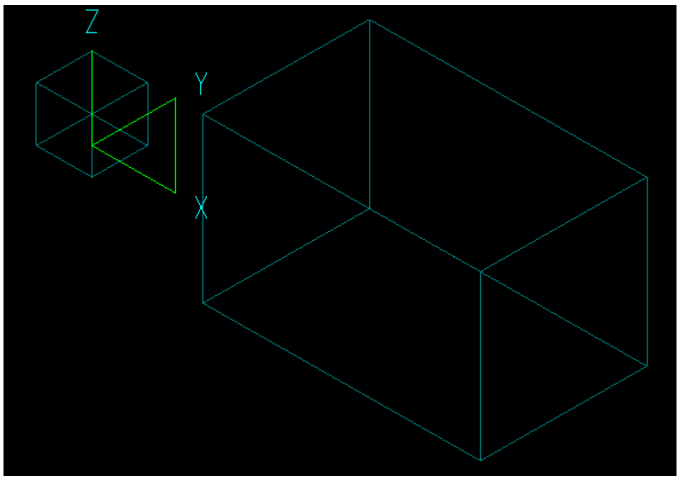


Picture 84

Picture 83 shows the result of executing **Box.ppm**. Unlike functions that use the new format, custom functions defined using the old format can produce a result.

File Main.ppm

```
Output(Box(1, 1, 1), Move(Box(5, 3, 3), 6, 0, 0));
```



Picture 85

Functions For Creating Tools (Supported by SymbolDesigner)

SymbolDesigner has the possibility to create tools that optimize script creation process. A tool is a special PPM custom function that gets script parameters and points input from the user and then generates scripts. The following functions allow requesting input parameters from the user.

Example of a tool:

```
TArc(ds, p1, R, AS, AE)
{
        ds = GetProperties("", "Style", "", "");
        p1 = GetPoint("Input arc center point", "p1", "", Point(0, 0, 0));
        R = GetDistance("Input arc radius", "Radius", "/L", 0.1);
        AS = GetAngle("Input arc start angle", "AStart", "/Ip1", 0.0);
        AE = GetAngle("Input arc end angle", "AEnd", "/Ip1", 270);
        ar = Arc(ds, R, AS, AE, p1);
        Output(ar);
}
```

GetAngle

Gets the angle identifier or the expression specified with the third parameter.

Call format:

```
<id> = GetAngle(sPrompt, sName, sToolOpt, DefExpr);
```

Arguments:

sPrompt - instruction text that will be displayed on the screen.
sName - point name.
sToolOpt - tool options.
DefExpr - default expression.

Example of using the function:

```
AS = GetAngle("Input arc start angle", "AStart", "/Ip1", 0.0);
```

GetDistance

Gets the distance identifier or the expression specified with the third parameter.

Call format:

```
<id> = GetDistance(sPrompt, sName, sToolOpt, DefExpr);
```

Arguments:

sPrompt - instruction text that will be displayed on the screen.
sName - point name.
sToolOpt - tool options.
DefExpr - default expression.

Example of using the function:

```
R = GetDistance("Input arc radius", "Radius", "/L", 0.1);
```

GetGraphic

Gets graphic's identifier or the expression specified with the third parameter.

Call format:

```
<id> = GetGraphic(sPrompt, sName, sToolOpt, DefExpr);
```

Arguments:

sPrompt - instruction text that will be displayed on the screen.
sName - point name.

sToolOpt - tool options.
DefExpr - default expression.

Example of using the function:
g = GetGraphic("Select 2D Entity", "g", "/nC/E", 0);

GetPoint
Gets the point identifier or the expression specified with the third parameter.

Call format:
<id> = GetPoint(sPrompt, sName, sToolOpt, DefExpr);

Arguments:
sPrompt - instruction text that will be displayed on the screen.
sName - point name.
sToolOpt - tool options.
DefExpr - default expression.

Example of using the function:
p1 = GetPoint("Input arc center point", "p1", "", Point(0,0,0));

GetSelected (Obsolete)
Gets the selected graphic identifier or the expression specified with the third parameter.

Call format:

<id> = GetSelected(sPrompt, sName, sToolOpt, DefExpr);

Arguments:
sPrompt - instruction text that will be displayed on the screen.
sName - point name.
sToolOpt - tool options.
DefExpr - default expression.

GetProperties
Gets property set identifier or the expression specified with the third parameter.

Call format:
<id> = GetProperties(sPrompt, sName, sToolOpt, DefExpr);

Arguments:
sPrompt - instruction text that will be displayed on the screen.
sName - point name.
sToolOpt - tool options.
DefExpr - default expression.

Example of using the function:
ds = GetProperties("", "Style", "/oT:dim", "");

Tool Options

Tool Option	Description
/L	use "from last point" (GetDistance(), GetAngle()).
/F	return distance or angle as value (not formula) (GetDistance(), GetAngle()), return point as value (Point(x, y)) even if ID (not formula) (GetPoint())

/I<id name>	use "from <id name> point" (GetDistance(), GetAngle())
/C	get point/entity array (GetPoint() / GetGraphic()) or repeat request any number of times (EditPoint())
/pF	set as first point (GetPoint())
/nC	do not calculate while tracking
/nE	get Entity as Not Editable (read only - ex. don't remove from output)
/U	deselect all before "Get..." argument
/E	get editable point (GetPoint())
/R	get point as reference point (GetPoint())
/N	goto next parameter while accepted
/oT:<type>	object type (GetProperties()). <type> can be one of: • pline - polyline (line, rectangle, polyline) • arc - arc (circle, ellipse, arc) • text - text • dim – dimension
/aF	show action(button) “Finish” (for point array)
/aB	show action(button) “Back one step” (for point array)
/aC	show action(button) “Close” (for point array)