Spezifikation

ODB
OFML Database*
(OFML Part I)

Version 2.4

Status: Release

Jochen Pohl, Ekkehard Beier, Sebastian Schmidt (EasternGraphics GmbH)†

January 6, 2022

†ODB was developed by EasternGraphics GmbH on behalf of industrial association Büro und Arbeitswelt e. V. (IBA).

Contents

1 Introduction

Using the ODB you can describe the geometric and, to a certain extent, the logical characteristics of planned objects. The ODB's goal is to have a descriptive form that can be easily written into a program and that can be checked for consistency. In order to achieve this, the ODB data are arranged in a table.

1.1 Survey of Tables

- Geometry–tables
  There are two separate tables for the 2D- and the 3D-geometries. They are described in the sections 2 and 3.
- Tables for attachment points
  Planning objects are placed in relation to other planning objects other using attachment points. The attachment points are defined in the ODB using three tables described in section 4.
- Function table
  In the table columns, arithmetic and logical expressions can often be used, which are formulated in Reversed Polish Notation. These expressions can reference predefined or user defined functions. The user defined functions are defined in the table described in section 5.
- Layer table
  3D layers are optionally defined in the table described in section 6.

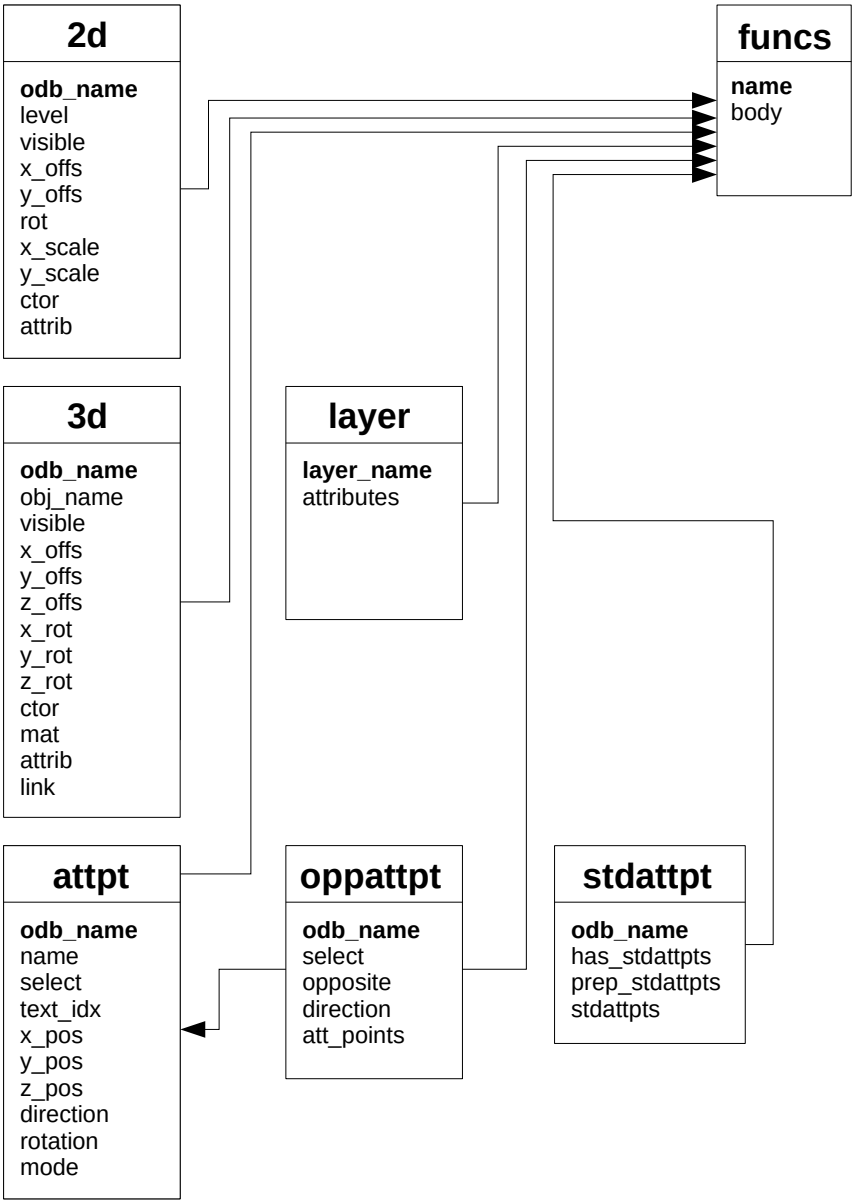


Figure 1: Survey of Tables

1.2 Regulations regarding the format

CSV tables (comma separated values) are used as the physical exchange format between OFML conform applications. The following regulations apply for this:

- Each of the tables described below is included in exactly one file. The file name is made of the prefix "`ocd_`", the specified table name and the suffix "`.csv`"; the table name is written completely in small letters.
- `ISO-8859-1` (Latin-1) is used as the character set.
- Each line of the file represents a data record[1].
  Blank lines, i.e. lines consisting of zero or several blank characters (0x20) or tabulators (0x09), are ignored. Lines starting with a number sign ('`#`'=0x23) are interpreted as a comment and are ignored, too.
- The representations of the individual fields of a data record are separated from each other by a semicolon ('`;`'=0x3B).
- The value of a field consists of zero or more printable characters from `ISO-8859-1` (0x20-0x7E, 0xA1-0xFF).
- The representation of a field is derived from the value of the field by replacing each quotation mark ('`"`'=0x22) by two quotation marks and enclosing the resulting string in quotation marks. If the value of a field does not start with a quotation mark and does not contain a semicolon ('`;`'=0x3B), the value itself (i.e. without any modifications) can be used as the field representation.

[1] A line is terminated either by an LF character (0x0A) or by a sequence of CR (0x0D) and LF.

2 2D-Geometries

Table name: `odb2d`
Obligatory table: yes

The 2D-geometry of an OFML object is described by one or more consecutive entries in the 2D-table. The purpose of each of these entries is to create a graphical primitive[2] and contains their scale, rotation, offset, and, if applicable, additional attributes, such as color, line width etc.

The structure of the 2D-geometry table is summarized in the 1 table and described in detail below.

field number	field name	description
1	`odb_name`	ODB name
2	`level`	hierarchy level
3	`visible`	visibility control
4	`x_offs`	X-offset
5	`y_offs`	Y-offset
6	`rot`	Rotation (around Z-axis)
7	`x_scale`	X-scale
8	`y_scale`	Y-scale
9	`ctor`	creating 2D objects
10	`attrib`	setting graphical attributes

Table 1: 2D geometries

2.1 ODB Name

Objects for which you want to create a 2D-geometry using the ODB, provide a fully qualified ODB name. It is comprised of the package name containing the used ODB and the basic ODB name, which determines the entries to be used for the 2D- and 3D-tables. An example of a fully qualified ODB name is `::foo::bar::BAZ`, where `::foo::bar` is the package name, and `BAZ` is the basic ODB name.

The 2D-table consists of a series of ODB blocks. An ODB block consists of several consecutive entries; the first entry in the column `odb_name` contains the basic ODB name, while in all following entries in this block, the column `odb_name` is blank.

2.2 Hierarchy Level

Within an ODB name, the entries in the 2D-table can be sorted by hierarchy. This enables you to group a variety of elements and transform them as a group (to scale, rotate, and offset.)

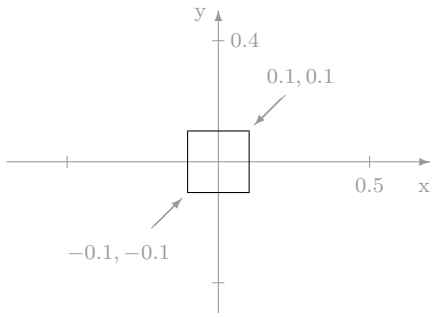
[2]When referencing external geometries, an entry can also create complex 2D geometries, which will be handled as whole objects.

The default hierarchy level for first entry of an ODB name is 0. If you need to combine several elements of a group, they list them as consecutive entries in the 2D-table and give them the same hierarchy level, which must be one level above the level of the entry determining the transformation of the group. The entry determining the transformation of a group is always the last entry before the group whose hierarchy level is lower than the hierarchy level of the group.

In the final form of the following example, four lines forming a square are combined into a group, where the entire group has an X-offset of 0.6 and a Y-offset of 0.4. The origin of the local coordinate system is located in the center point of the non-rotated square. Furthermore, the square is contained in a rectangle with the dimensions 1.2×0.8 so that the center points of the square and the rectangle are identical.

In the first step, the square consists of four lines, so that the center point of the square is identical with the origin of the coordinate system of the OFML object:

odb_name	level	visible	offs		rot	scale		ctor	attrib
			x	y		x	y		
BAZ	0		−0.1	−0.1	0.0	0.2	1.0	hline	
	0		−0.1	0.1	0.0	0.2	1.0	hline	
	0		−0.1	−0.1	0.0	1.0	0.2	vline	
	0		0.1	−0.1	0.0	1.0	0.2	vline	



In the example above, four lines are created from top to bottom: from $-0.1, -0.1$ to $0.1, -0.1$, from $-0.1, 0.1$ to $0.1, 0.1$, from $-0.1, -0.1$ to $-0.1, 0.1$, and from $0.1, -0.1$ to $0.1, 0.1$. For detailed information on creating lines using the functions `hline` and `vline`, refer to section 2.7.1.

In the next step, the lines are combined into a group without moving the group yet. The lines are still where they were in the above illustration.

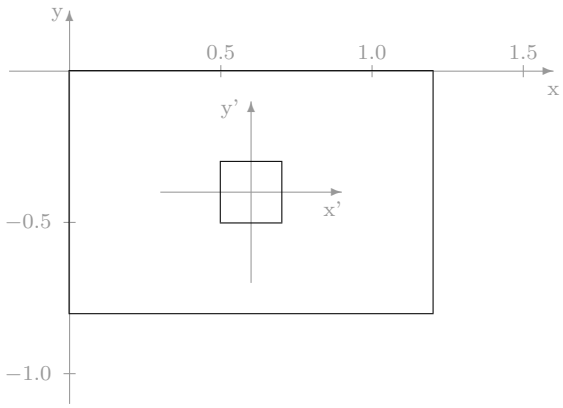
odb_name	level	visible	offs		rot	scale		ctor	attrib
			x	y		x	y		
BAZ	0		0.0	0.0	0.0	1.0	1.0		
	1		−0.1	−0.1	0.0	0.2	1.0	hline	
	1		−0.1	0.1	0.0	0.2	1.0	hline	
	1		−0.1	−0.1	0.0	1.0	0.2	vline	
	1		0.1	−0.1	0.0	1.0	0.2	vline	

You can see that the lines are merely preceded by a blank object[3] with the hierarchy level 0, and the hierarchy level of the lines is consequently raised to 1. This is how the lines are created in relation to the object in the table's first line.

[3]Objects with a blank `ctor` column are not graphically represented.

In the first step, the group is offset by 0.6 in X-direction, and by −0.4 in Y-direction, and at the same time, a rectangle with the dimensions 1.2 (width) and 0.8 (height) is added. Its upper left corner is located at the origin of the coordinate system of the OFML object. The following figure shows the group's moved local coordinate system with the axis names x' and y'.

odb_name	level	visible	offs		rot	scale		ctor	attrib
			x	y		x	y		
BAZ	0		0.6	−0.4	0.0	1.0	1.0		
	1		−0.1	−0.1	0.0	0.2	1.0	hline	
	1		−0.1	0.1	0.0	0.2	1.0	hline	
	1		−0.1	−0.1	0.0	1.0	0.2	vline	
	1		0.1	−0.1	0.0	1.0	0.2	vline	
	0		0.0	0.0	0.0	1.2	−0.8	quadrat	



The table illustrates how the upper level object of a moved group must contain the offset for this group, in this case in the first row of the table. The rectangle is created in the last row of the table, in which a default square with the dimensions 1.0×1.0 is scaled in X-direction to 1.2 and in Y-direction to −0.8. For detailed instructions on how to create rectangles, refer to section2.7.2.

2.3 Visibility

In some cases, you may want to display parts of a 2D-symbol only for certain configurations of the OFML object on which the symbol is based. The visibility can be controlled by an entry in the `visible` column. The entry is displayed when the `visible` column is blank or when it contains a value other than 0. If the value in the `visible` column is 0, neither the entry nor any existing lower hierarchy entries are displayed.

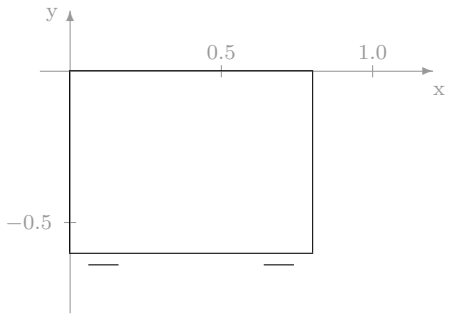
In the following example, we want to display a short line (representing a door handle) to the left or right in front of a wardrobe (represented by a rectangle). The decision is based on the value of the parameter `$HANDLE`, based on the OFML object and is either `"L"` for left or `"R"` for right.

In the first table, a wardrobe is displayed with the dimensions 0.8 (width) and 0.6 (depth) and two handles, one on the right, and one on the left. The lines symbolizing the handles are 0.1 long. They start or end at a distance of 0.05 from the left or right edge of the wardrobe, and they are offset 0.3 forward from the wardrobe.

In this case, the symbol representing the wardrobe is created differently than in section 2.2. While in section 2.2, the rectangle was not moved and had to be scaled negative in Y-direction in order to flip it down; in this example, the origin of the rectangle is being moved to its lower left corner so that it can be scaled positive in Y-direction.

The second row in the table represents the left handle, and the third row represents the right handle. Since the origin (the starting point) of the line representing the right handle was indicated as its end point, the line must be scaled negative in X-direction in order to place its end point left of its starting point.

odb_name	level	visible	offs		rot	scale		ctor	attrib
			x	y		x	y		
CUPBOARD	0		0.0	−0.6	0.0	0.8	0.6	quadrat	
	0		0.05	−0.63	0.0	0.1	1.0	hline	
	0		0.75	−0.63	0.0	−0.1	1.0	hline	



In order select the handles to display, the last two rows of the column `visible` must be filled. However, we cannot use a constant value as in the previous examples. Also, the parameter `$HANDLE` cannot be used because its value is a string, while the expected result in the `visible` column is a number. Therefore, in order to display the handle, we need an expression with a result of 1.0 for the left handle and the right handle, and a result of 0.0 in other cases. The expression for the left handle in Reverse Polish Notation is "'`$HANDLE "L" ==`"', and for the left handle it is "'`$HANDLE "R" ==`"' accordingly.

Since the `visible` column has a limited field width, the expressions are typically not entered directly in the column; they are written as a function. The function is saved in a separate table, as described in section 5. The two following tables show the relevant entries in the 2D-table and in the function table.

odb_name	level	visible	offs		rot	scale		ctor	attrib
			x	y		x	y		
CUPBOARD	0		0.0	−0.6	0.0	0.8	0.6	quadrat	
	0	GL	0.05	−0.63	0.0	0.1	1.0	hline	
	0	GR	0.75	−0.63	0.0	−0.1	1.0	hline	

name	body
GL	$HANDLE "L" ==
GR	$HANDLE "R" ==

2.4 Offset

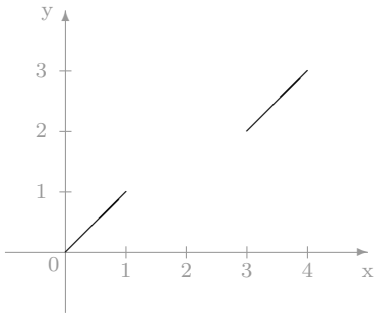
Every object you want to add has an insertion point that is used to place it in the origin of a coordinate system. This insertion point is always the origin of the coordinate system

in which the object's coordinates were captured. Using the offset parameter, the insertion point can be moved in X-direction and the Y-axis. The object will be moved after scaling and rotating the added object, if applicable.

This offset can be demonstrated using a diagonal line. A diagonal line that has been added using the `dline` command in the `ctor` column extends from point $0.0, 0.0$ (the insertion point) to point $1.0, 1.0$. If this diagonal line is not moved, it extends from $0.0, 0.0$ to $1.0, 1.0$ in the coordinate system of the OFML object as well. If you want to move it to extend from $3.0, 2.0$ to $4.0, 3.0$, you need to move its insertion point 3.0 in X-direction and 2.0 in Y-direction.

The following table and the corresponding figure show two diagonal lines being added; the first line is not offset, and the second is offset by $3.0, 2.0$.

odb_name	level	visible	offs		rot	scale		ctor	attrib
			x	y		x	y		
FOO	0		0.0	0.0	0.0	1.0	1.0	dline	
	0		3.0	2.0	0.0	1.0	1.0	dline	

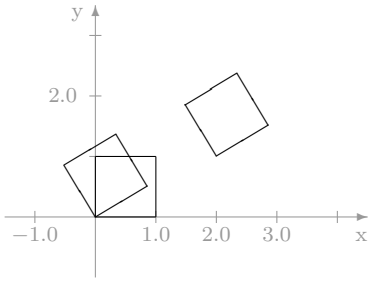


2.5 Rotation

The rotation parameter is used to rotate added objects around the origin of their local coordinate system. The rotation angle is indicated as a mathematically positive value (counter-clockwise) in degrees. The object is rotated after completion of scaling and offset, if applicable.

The following example shows three squares. Every square created using `quadrat` in the `ctor` column has a side length of 1.0 before scaling. The first square is added without rotation or offset. The second square is rotated 30 degrees. The third square is rotated 30 degrees and then offset 2.0 in X-direction and 1.0 in Y-direction.

odb_name	level	visible	offs		rot	scale		ctor	attrib
			x	y		x	y		
FOO	0		0.0	0.0	0.0	1.0	1.0	quadrat	
	0		0.0	0.0	30.0	1.0	1.0	quadrat	
	0		2.0	1.0	30.0	1.0	1.0	quadrat	



2.6 Scaling

Scaling, next to offsetting, is the most important transformation for the creation of 2D-geometries using the ODB. The reason for this is that many graphical primitives are created in the shape of "unit primitives." The primary feature of the unit primitives is that for all corner and end points, x_i, y_i applies, since x_i as well as y_i are either 0.0 or 1.0. By scaling in X- and Y-direction, the unit primitives can be sized as desired. Since their coordinate values are either 0.0 or 1.0 their dimension in X- or Y-direction can generally be used as scaling factor.

In addition, objects can be mirrored using scaling parameters. If, for instance, the value in the `x_scale` column is set to -1.0 the object is mirrored at the Y-axis of its local coordinate system. Likewise, a -1.0 in the `y_scale` column mirrors the object along the X-axis of its local coordinate system.

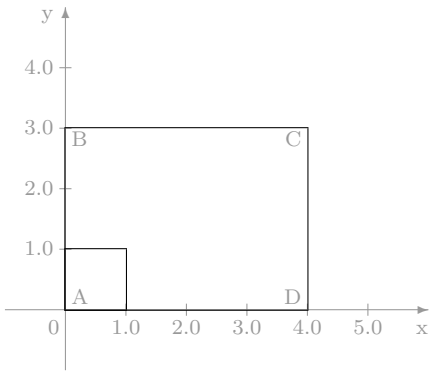
Finally, scaling and mirroring can be combined in one scaling factor. If, for instance, you want to increase an object in size in X-direction by 2.5 and mirror it along the Y-axis at the same time, enter -2.5 in the `x_scale` column.

The scaling will be performed on the object before any rotation or offsetting.

To ensure that an object will not be scaled in a certain direction, enter the scaling factor 1.0. The value 0.0 as scaling factor is not permissible.

In the following example, a "unit square" is created in the first row. The same square is created in the second row, though this one is scaled 4.0 in X-direction and 3.0 in Y-direction. The following figure shows both squares.

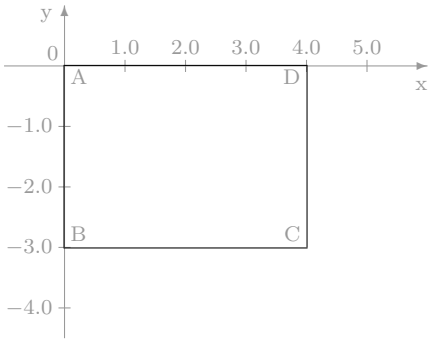
odb_name	level	visible	offs		rot	scale		ctor	attrib
			x	y		x	y		
FOO	0		0.0	0.0	0.0	1.0	1.0	quadrat	
	0		0.0	0.0	0.0	4.0	3.0	quadrat	



In the next example, the same rectangle as the one in the second line of the previous example is created, the only difference being that it is mirrored along the X-axis by negating the scaling factor. You can recognize that the object is mirrored by looking at the letters indicating the corner points of the rectangle.

In this case, the same effect could have been achieved without mirroring, by offsetting the rectangle by -3.0 in Y-direction. The effect is different, however, when the mirrored object is not symmetric to the mirroring axis.

odb_name	level	visible	offs		rot	scale		ctor	attrib
			x	y		x	y		
FOO	0		0.0	0.0	0.0	4.0	−3.0	quadrat	



2.7 Creating Objects

The `ctor` is used to create 2D-objects. There are generally three different scenarios:

- **The column is blank.** In this case the 2D-object will not be graphically represented. This can be useful when you want to create a group of objects, as described in section 2.2, and you want their transformation to be determined by the object at the next higher level in the hierarchy.
- **This column directly creates an object.** The available objects are vertical, horizontal and diagonal lines, squares, circles and arcs, ellipses, points, text and stretch.
- **The column references an external geometry.** In this case, it refers to a file that may contain a complex 2D-geometry.

If it is not blank, the `ctor` column always contains a function call in Reverse Polish Notation causing the creation of a 2D-object.

The `ctor` column can also contain a function defined in the function table. This function, as well as functions indirectly or directly invoked by this function, can all invoke the following functions.

2.7.1 Lines

Three different functions are used in the `ctor` column to create "unit lines". These functions are listed in the 2 table.

function	start point	end point
`hline`	$x = 0.0; \quad y = 0.0$	$x = 1.0; \quad y = 0.0.$
`vline`	$x = 0.0; \quad y = 0.0$	$x = 0.0; \quad y = 1.0.$
`dline`	$x = 0.0; \quad y = 0.0$	$x = 1.0; \quad y = 1.0.$

Table 2: Functions for line creation

The `dline` function helps you create diagonal lines. By using appropriate scaling you can create lines of any length and slope. The exceptions are horizontal and vertical lines, since they would need to be created using a scaling factor of 0.0, which is not permissible. Therefore, horizontal and vertical lines need to be created using `hline` and `vline`.

In order to display a line with the starting point x_0, y_0 and the end point $x_1, y_1$the parameters x_{offs}, y_{offs} for offset, and x_{scale}, y_{scale} for scaling (as shown in table 3) can be calculated.

condition	function	x_{offs}	y_{offs}	x_{scale}	y_{scale}
$y_0 = y_1$	`hline`	x_0	y_0	$x_1 - x_0$	1.0
$x_0 = x_1$	`vline`	x_0	y_0	1.0	$y_1 - y_0$
in other cases	`dline`	x_0	y_0	$x_1 - x_0$	$y_1 - y_0$

Table 3: Calculating offset and scaling for lines

2.7.2 Squares and Rectangles

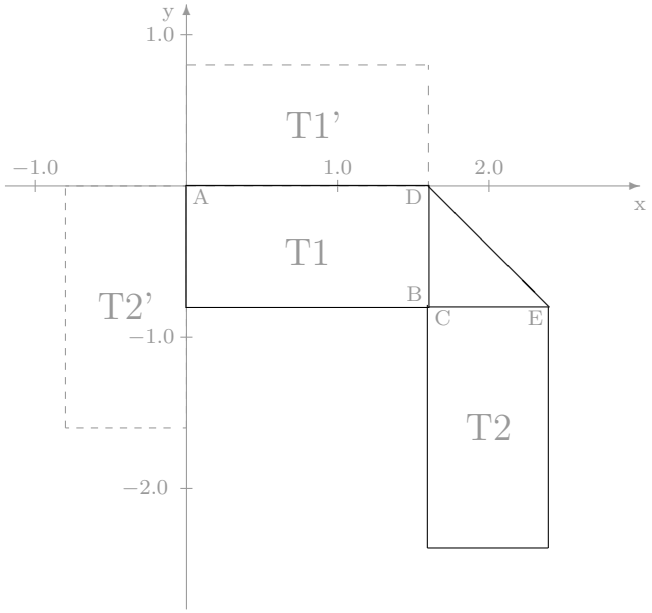
You can create squares and rectangles using the `quadrat` function in the `ctor` column. The `quadrat` function creates a square with a side length of 1.0, whose left lower corner is located at the origin of the local coordinate system. To create a rectangle with a width of w and a height of h, set the scaling factor in X-direction to w and the scaling factor in Y-direction to h. The rectangle created using this method can now be rotated and moved as desired.

The following example shows two tables represented by rectangles. These tables are positioned in a 90-degree angle to each other and are linked by a connector. Both tables are 1.6 wide and 0.8 deep. The first table is positioned horizontally with its upper left corner (A) at the origin of the OFML object's coordinate system. The lower left corner (C) of the second table, which is rotated by 90 degrees, is located at the first table's lower right corner (B). A

connecting board is located between the tables, symbolized by a line between the first table's upper left corner (D) and the second table's upper left corner (E)[4].

In the first row, the table (T1) is created in a horizontal position. This is trivial since the unit square merely needs to be scaled correctly in X- and Y-direction with a negative scaling factor in Y-direction in order to mirror the table on its X-axis[5]. In the second row, the table in vertical position, T2, is created. It is scaled using the same method as the table in horizontal position, then turned 90 degrees in clockwise direction[6] and then moved into position. In the third row, the line between the two tables is drawn.

odb_name	level	visible	offs		rot	scale		ctor	attrib
			x	y		x	y		
FOO	0		0.0	0.0	0.0	1.6	−0.8	quadrat	
	0		2.4	−0.8	−90.0	1.6	−0.8	quadrat	
	0		1.6	0.0	0.0	0.8	−0.8	dline	



2.7.3 Circles, Arcs and Ellipses

The three functions listed in the 4 table are used to draw circles, arcs and ellipses.

For all three functions, the center point is always located at the origin of the coordinate system. For arcs (`arc`) the arc runs mathematically positive (counter-clockwise) from the start to the end angle.

The following simple example shows a round table with a diameter of 1.2 and a center point located at $0.6, -0.6$. There is no illustration for this example.

[4]This example is not particularly relevant for practice since typically, the tables would be handled separately.

[5]In the illustration, the non-mirrored table, T1', is shown with dashed lines.

[6]The rotated table, T2', is shown with dashed lines.

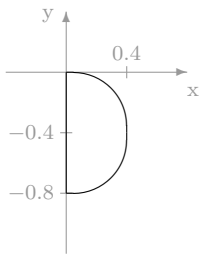
syntax	x–radius	y-radius	start angle	end angle
`circle`	1.0	1.0	0.0	360.0
α_{start} α_{end} `arc`	1.0	1.0	α_{start}	α_{end}
x_{radius} y_{radius} `ellipse`	x_{radius}	y_{radius}	0.0	360.0

Table 4: Commands for circles, arcs and ellipses

odb_name	level	visible	offs		rot	scale		ctor	attrib
			x	y		x	y		
FOO	0		0.6	−0.6	0.0	0.6	0.6	circle	

The following example demonstrates the use of arcs. You may want to create a semicircular table with a radius of 0.4. The table's straight side intersects the center point of the semicircle and is located on the negative Y-axis.

odb_name	level	visible	offs		rot	scale		ctor	attrib
			x	y		x	y		
FOO	0		0.0	−0.4	0.0	0.4	0.4	-90.0 90.0 arc	
	0		0.0	0.0	0.0	1.0	−0.8	vline	



Since ellipse arcs are currently not supported, they are drawn using scaled arcs[7]. Please note, though, that the start and end angles typically change with a difference in scaling in X- and Y-direction. You can calculate the angle for the arc using the equation

$$\alpha_{kreis} = \arctan\left(\frac{y_{scale}}{x_{scale}} \tan \alpha_{ellipse}\right)$$

, making sure you use the appropriate quadrant.

2.7.4 Points

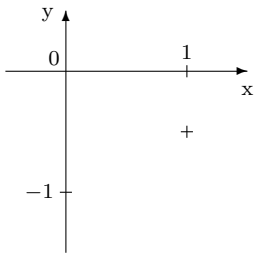
Use the `point` function to create a single point at the origin of the coordinate system. Rotation and scaling are not considered for a point object[8].

IN the following example, a point is placed at 1.0, −0.5. In the illustration the point is shown as a small cross for better visibility. In reality, it the point displays as a placed pixel.

[7] We do not recommend drawing ellipses by scaling circles because several snap modes, especially the perpendicular snap mode will cease to function properly due to the difference in scaling in X- and Y-direction.

[8] If rotation and/or scaling are indicated for a point object, and if there are objects below the hierarchy level of the point object, these objects are influenced by the indicated rotation and/or hierarchy.

odb_name	level	visible	offs		rot	scale		ctor	attrib
			x	y		x	y		
FOO	0		1.0	−0.5	0.0	1.0	1.0	point	



2.7.5 Text

The `text` function creates a text object with a reference point at the origin of the local coordinate system. The parameters should indicate the orientation of the text prior to its rotation in relation to the reference point, as well as the text itself. The default character size for uppercase text excluding descenders is 0.1.

The first parameter of the `text` function determines the horizontal orientation of the text prior to its rotation. Valid values for the parameter are found in the 5 table[9]. The text is oriented vertically in a way that the base line[10] of the text intersects the reference point.

value	meaning
−1.0	The text is always located to the immediate right of the reference point.
0.0	The text is centered horizontally to the reference point.
1.0	The text is located to the immediate left of the reference point.

Table 5: Orienting text objects

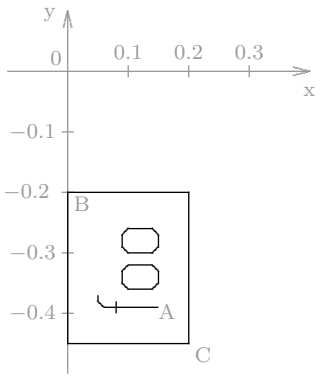
Using scaling factors to control the text height and width is not recommended. Although this may work well with the current vector-based text implementation, it might cause problems if bitmap fonts are introduced in the future. Instead, use the `fheight` and `faspect` attribute functions to control the text height and extension. These functions are described in section 2.8.

The following example shows vertically oriented text, readable from the right. The text is created left-aligned to the reference point (A) with the coordinates $0.15, -0.4$. Since the orientation is indicated for the non-rotated text, the text is located above the reference point. The text has a frame with an upper left corner (B) at $0.0, -0.2$ and a lower left corner (C) at $0.2, -0.45$.

odb_name	level	visible	offs		rot	scale		ctor	attrib
			x	y		x	y		
FOO	0		0.15	−0.4	90.0	1.0	1.0	`-1.0 "foo" text`	
	0		0.0	−0.45	0.0	0.2	0.25	quadrat	

[9] All floating point values are allowed for the orientation. For positive values, the center point of the text is always located to the left of the reference point, and its distance from the reference point grows with growing absolute values. The same applies to negative values, except that the center point of the text is located to the right of the reference point.

[10] The base line is the line on which upper case characters sit-excluding descenders.



2.7.6 Stretch

The function *len a b c* `stretch` doesn't create an object but changes the geometries of the objects below it's hierarchy level.

The parameters are defined as follows:

The parameter *len* specifies the lenght of the segment to insert, negative values are permitted and are interpreted as contraction.

The parameters *a b c* describe the cutting line in the form $ax + by = d$. The vector *a b* is the normal vector of the line, *d* is the distance of the line to the origin of the coordinate system.

The following example shows an object which is stretched by 0.7 units along the *y*-axis.

odb_name	level	visible	offs		rot	scale		ctor	attrib
			x	y		x	y		
FOO	0		0.0	0.0	0.0	1.0	1.0	`0.7 0 1 0.25 stretch`	
	1		0.0	0.0	0.0	1.0	1.0	`"foo" egms`	

2.7.7 External Geometries

If you want to draw more complex geometries that you don't want scaled as whole objects, it may make sense to save them as EGM symbol file and to integrate them in the ODB as external geometry[11].

The integration function is called `egms`. The expected argument for the function is a string containing the name of the external geometry. This name can be either fully qualified or not qualified. In the latter case, the system will try to find the geometry in the package containing the ODB.

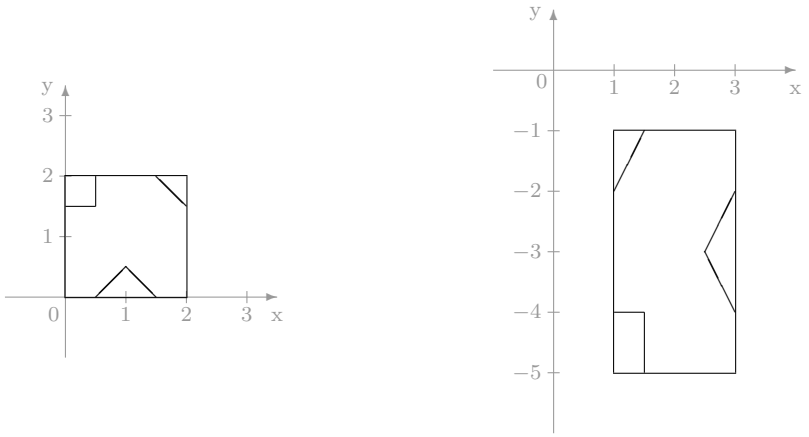
The coordinates in the external geometry are interpreted in the local coordinate system of the ODB entry referencing the external geometry. They can subsequently be transformed by scaling, rotating or moving.

Currently, you cannot set attributes for external geometries.

[11]This is a standard procedure to create geometries in an ODB created from a FOS format conversion. When initially saving ODB-based geometries, the use of external geometries is not recommended since the use of graphical primitives directly supported by ODB is more efficient.

The following example references an EGM symbol. The EGM symbol is shown in the illustration on the left in its local coordinate system. The resulting geometry in the coordinate system of the OFML is shown in the illustration on the right. The name of the EGM symbol is `bar`. Since the name is not qualified, the package containing the ODB should also contain a file with the name `bar.egms`.

odb_name	level	visible	offs		rot	scale		ctor	attrib
			x	y		x	y		
FOO	0		3.0	−5.0	90.0	2.0	1.0	"bar" egms	



2.8 Attributes

In the 2D table's `attrib` column, you can set several attributes valid for the 2D-object that was created for the respective entry.

The `attrib` column can also contain a function defined in the function table. This function, as well as functions they may invoke, can invoke any of the following functions.

2.8.1 Color

The color-setting function is called `col`. It expects three arguments in the range from 0.0 through 1.0, specifying the red, green and blue values of the color. If no color is set for an object, that object is displayed black.

Color can be set for all graphical primitives. Graphical primitives are `hline`, `vline`, `dline`, `quadrat`, `circle`, `arc`, `ellipse`, `point` and `text`.

The following example shows how a red rectangle with two blue diagonals and the dimensions 2.0×1.0 is created. The rectangle's upper left corner is located at the origin of the OFML object's coordinate system, and the lower right corner is located at $2.0, -1.0$.

odb_name	level	visible	offs		rot	scale		ctor	attrib
			x	y		x	y		
FOO	0		0.0	−1.0	0.0	2.0	1.0		
	1		0.0	0.0	0.0	1.0	1.0	quadrat	1.0 0.0 0.0 col
	1		0.0	0.0	0.0	1.0	1.0	dline	0.0 0.0 1.0 col
	1		0.0	1.0	0.0	1.0	−1.0	dline	0.0 0.0 1.0 col

2.8.2 Line Width

The function for setting the line width is called `lwidth`. The argument it expects is a positive number specifying the line width in pixels. The standard line width is 1 pixels.

The line width can be set only for the primitives `hline`, `vline`, `dline`, `quadrat`, `circle`, `arc`, and `ellipse`.

The following example shows how to create an ellipse with a center point at $1.0, -0.5$, a radius of 1.0 in X-direction and a radius of 0.5 in Y-direction. The line width is 2 pixels.

odb_name	level	visible	offs		rot	scale		ctor	attrib
			x	y		x	y		
FOO	0		1.0	−0.5	0.0	1.0	1.0	1.0 0.5 ellipse	2 lwidth

2.8.3 Line Style

The function for setting the line style is called `lstyle`. The arguments it expects are two numbers indicating the line pattern and extension factor. Lines are solid by default, or dashed when the object they are associated with is selected.

The line style can be set for the primitives `hline`, `vline`, `dline`, `quadrat`, `circle`, `arc`, and `ellipse`.

The first argument is the line pattern, and it can accept the values in the 6 table . The second argument is a factor whose value is interpreted as number of pixels. The exact meaning of the factor depending on the used pattern is also described in the 6 table[12].

value	description
−1	Using a predefined line style.
0	Drawing a solid line.
1	Drawing a dashed line. The factor determines the length of the displayed and non-displayed line segments.
2	Drawing a dotted line. The factor determines the distance between the center points of two neighboring points.
3	Drawing a point-dot line. The factor determines the length of the displayed line segment and the half length of the non-displayed segments.
4	Drawing a dashed double-dotted line. The factor determines the length of the displayed line segment and a third of the length of the non-displayed segments.
5	Drawing a dashed triple-dotted line. The factor determines the length of the displayed line segment and a quarter of the non-displayed segments.

Table 6: Line Pattern

When using line patterns, please be aware that the display of selected object is based on dashed lines. This applies only to lines that were not explicitly assigned a line pattern

[12]The actual line segment length can differ from the information in the 6 table, depending on the driver used for the 2D-version. This means that especially the OpenGL driver provides very limited possibilities, while the X11 driver provides rather exact results considering the abilities of a pixel display.

differing from −1. Therefore, when creating 2D-symbols, make sure you do not assign a line pattern to all lines of the symbol.

In the following example, a dashed rectangle with dotted diagonals is created using the same dimensions as the example in section 2.8.1. The extension factor is 4 in all cases, which typically provides good visual results.

odb_name	level	visible	offs		rot	scale		ctor	attrib
			x	y		x	y		
FOO	0		0.0	0.0	0.0	2.0	−1.0	quadrat	1 4 lstyle
	0		0.0	0.0	0.0	2.0	−1.0	dline	2 4 lstyle
	0		0.0	−1.0	0.0	2.0	1.0	dline	2 4 lstyle

2.8.4 Point Size

The diameter of a point is set using the function `psize`. It expects an argument indicating the point size in pixels.The standard size of a point is 1 pixels.

The point size can be set for objects created with the `point` function.

The following example is identical to the one in section 2.7.4, except that the point is displayed with a diameter of 5.

odb_name	level	visible	offs		rot	scale		ctor	attrib
			x	y		x	y		
FOO	0		1.0	−0.5	0.0	1.0	1.0	point	5 psize

2.8.5 Font Height

For objects created with the function `text`, use the `fheight` function to set the font height.

The `fheight` function expects a floating point argument indicating the font height in units of the local coordinate system. The font height is the height of an uppercase character excluding descenders. The standard height is 0.1.

2.8.6 Font Aspect

The function `faspect` determines the aspect of the used fonts for objects created using the `text` function.

The `faspect` function expects a floating point argument indicating the font aspect. The standard value for the aspect is 1.0. A value between 0.0 and 1.0 decreases the character size in X-direction, and a value greater than 1.0 increases the character size in X-direction. An aspect value of 0.0 is not permissible. For negative values the response is not defined.

2.8.7 Layer

With the function `layer` each object can be assigned to a layer (see section 6). The function expects a string argument containing the name of the layer.

3 3D Geometries

Table name: `odb3d`
Obligatory table: yes

The 3D geometry of an OFML object is described by one or more successive entries in the 3D table. The purpose of each of these entries is to create a graphical primitive[13] and contains their position, rotation and other attributes, such as materials, selectability etc.

The structure of the 3D geometry table is summarized in table 7 and described in detail below.

field number	field name	description
1	`odb_name`	ODB name
2	`obj_name`	object name
3	`exist`	creation control
4	`x_offs`	x-offset
5	`y_offs`	y-offset
6	`z_offs`	z-offset
7	`x_rot`	x-rotation
8	`y_rot`	y-rotation
9	`z_rot`	z-rotation
10	`ctor`	3D object creation
11	`mat`	material(s) assignment
12	`attrib`	setting graphical attributes
13	`link`	reserved for future use

Table 7: 3D geometries

3.1 ODB Name

Objects for which you want to create a 3D-geometry using the ODB, provide a fully qualified ODB name. The name is comprised of the package name containing the used ODB and the basic ODB name, which determines the entries to be used for the 2D and 3D tables. An example of a fully qualified ODB name is `::foo::bar::BAZ`, where `::foo::bar` is the package name, and `BAZ` is the basic ODB name.

The 3D-table consists of a series of ODB blocks. An ODB block consists of several consecutive entries, of which the first in the column `odb_name` contains the basic ODB name, while for all following entries in this block, the column `odb_name` is blank.

3.2 Creating Objects

In order to create an object, you need to indicate a relative name that refers to the OFML object on the higher hierarchy level. The following rules apply to object names:

[13] When referencing external geometries, an entry can also create complex 3D geometries, which will be handled as whole objects.

- Within an ODB block, a name may be given out only once[14].
- There can be no hierarchical assignment within one name. To this extent, in general a name consists of linked basic names using the point (`.`) as a linking operator.
- If the name of an object implies the existence of a hierarchical predecessor, the successor must define the predecessor in the table.
- As a convention, the basic name consists of the prefix `o`, followed by an integer. For each given predecessor, this number begins at 1 for the first successor and is incremented accordingly for the following successors.

The following (incomplete) example shows how four objects are created. At the highest level–corresponding to Level 0 in the 2D ODB–two objects are created and named `o1` and `o2` based on the convention above. The other two objects' features are to be defined in relation to object `o2`. For this reason, a hierarchy level referring to `o2` is introduced, and accordingly, their names will be `o2.o1` and `o2.o2`.

odb_ name	obj_ name	ex ist	offs x	offs y	offs z	rot x	rot y	rot z	ctor	mat	attrib	link
BAZ	o1											
	o2											
	o2.o1											
	o2.o2											

3.3 Controlling the Creation Process

The process of creating an object can be controlled using the `exist` column. The object described in the row is created when the `exist` column is blank or the expression contained (typically a function) provides a numerical value other than 0. Otherwise, the object will not be created.

An object will also not be created if one of its hierarchical predecessors is not created based on an entry in the `exist` column.

3.4 Offset

Every added object has a unique attachment point that is always located at the origin of the local coordinate system. For a cube this is its left lower back corner, for a sphere it is its center point. When creating an object, this attachment point is located at the origin of the predecessor's coordinate system. Using the offset parameter, the attachment point can be offset in all three directions in relation to the attachment point.

When you move an object using this method, all successors are moved accordingly.

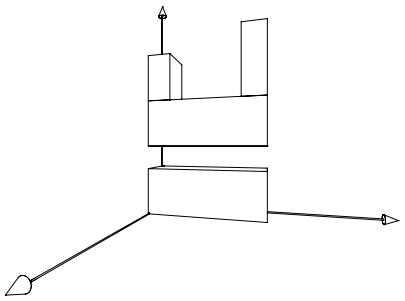
Offsetting always occurs independently from a possible rotation.

In the following example, `o1` is not offset from the OFML object's coordinate system. The origin of `o2` in relation to the OFML object's coordinate system is at $(0.0, 3.0, 0.0)$. The origins

[14] An ODB block is comprised of all entries under one ODB name.

of o2.o1 and o2.o2 in relation to the OFML object's coordinate system are at $(0.0, 0.5, 0.0)$ and $(0.4, 0.5, 0.0)$.

odb_ name	obj_ name	ex ist	offs			rot			ctor	mat	attrib	link
			x	y	z	x	y	z				
BAZ	o1		0.0	0.0	0.0	0.0	0.0	0.0	0.5 0.2 0.2 block			
	o2		0.0	0.3	0.0	0.0	0.0	0.0	0.5 0.2 0.2 block			
	o2.o1		0.0	0.2	0.0	0.0	0.0	0.0	0.1 0.2 0.2 block			
	o2.o2		0.4	0.2	0.0	0.0	0.0	0.0	0.1 0.3 0.2 block			



3.5 Rotation

When you indicate rotation angles that are not equal 0.0, you can rotate an object out of its orientation predefined by its type and give it a new orientation in reference to the OFML object or its predecessor.

A (x, y, z) rotation will be shown on basic rotations as follows:

1. x rotation in reference to the initial X-axis
2. y rotation in reference to the Y-axis of the coordinate system after step 1
3. z rotation in reference to the Y-axis of the coordinate system after step 2

Be aware that when you indicate several rotation angles not equal 0.0, there will be some interaction between the basic rotations.

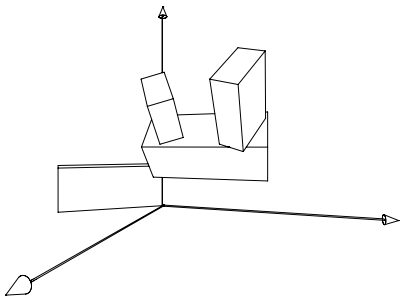
Note. When using hierarchy levels where each hierarchy level has exactly one assigned basic rotation, you may determine the processing sequence for the basic rotations.

Rotation angles are indicated in degrees and in mathematically positive sense, i.e. counter-clockwise.

When you rotate an object using this method, all successors are rotated accordingly.

Rotating always occurs independently from any possible offset.

odb_ name	obj_ name	ex ist	offs x	offs y	offs z	rot x	rot y	rot z	ctor	mat	attrib	link
BAZ	o1		0.0	0.0	0.0	0.0	−143.2	0.0	0.5 0.2 0.2 block			
	o2		0.0	0.3	0.0	0.8	0.0	0.0	0.5 0.2 0.2 block			
	o2.o1		0.0	0.2	0.0	0.0	22.9	0.0	0.1 0.2 0.2 block			
	o2.o2		0.4	0.2	0.0	0.0	−22.9	0.0	0.1 0.3 0.2 block			



3.6 Creating Objects

The `ctor` is used to create 3D-objects. This is done by invoking one of the following functions and providing the necessary parameters.

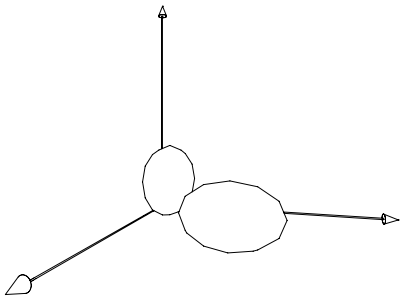
Note. For all of the primitives introduced in the following section, the origin of the local coordinate system is part of the local volume limit.

3.6.1 Ellipsoid

The `ellipsoid` function creates a homogeneous ellipsoid beginning at the origin of the local coordinate system and extending to all sides in accordance with the three radiuses.

The parameters are set by defining three radiuses: x, y, and z.

odb_ name	obj_ name	ex ist	offs x	offs y	offs z	rot x	rot y	rot z	ctor	mat	attrib	link
BAZ	o1		0.1	0.15	0.2	0.0	0.0	0.0	0.1 0.15 0.2 ellipsoid			
	o2		0.4	0.1	0.6	0.0	0.0	0.0	0.4 0.1 0.6 ellipsoid			



3.6.2 Import

The `imp` function imports an external 3D-record in order to create a corresponding primitive. The following formats are supported:

1. 3DS

   3DS is a binary format describing geometries based on triangle lists. Other 3DS file components are material, lighting, and camera data, though these components are ignored during the input.

   A 3DS record is imported into the ODB in a way that the minimal coordinate of its orthogonal volume limit is matched with the local coordinate system's origin.

   The file extension for 3DS files is `.3ds`.

2. OFF

   OFF is a simple ASCII format to describe indexed polygonal objects. In addition to the geometry file with the extension `.geo`, the ODB runtime environment optionally supports files with the extension `.vnm`, in which normal vectors can be assigned to vertices.

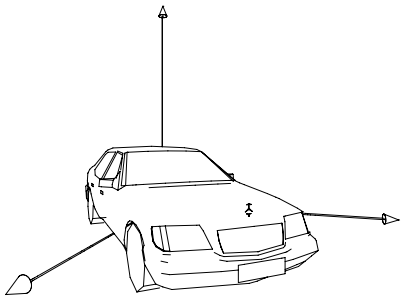
In general, the input records for the ODB should be unilateral and should describe closed bodies. The basic planes (triangles or polygons) should be simple, planar, convex and clockwise oriented.

Each record optionally supports a resolution reducing variant. This record's files differ from the primary record in that they are preceded by an underscore.

The first function parameter is the optionally fully qualified name of the record without the file extension. If the name is not or not fully qualified, it will be preceded by the qualifier of the fully qualified ODB name. Therefore, the record must be contained in the same package with the ODB.

The geometry is scaled using the following parameters in the sequence x, y, and z.

odb_ name	obj_ name	ex ist	offs			rot			ctor	mat	attrib	link
			x	y	z	x	y	z				
BAZ	o1		0.0	0.0	0.0	0.0	−50.1	0.0	"w140" 0.2 0.2 0.2 imp			

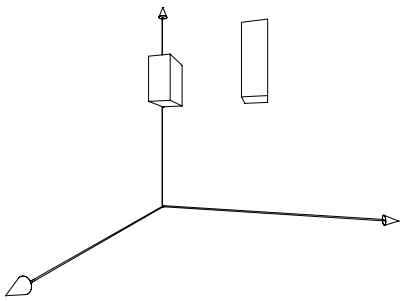


3.6.3 Top

The `top` function creates an invisible object generally used in order to combine objects and place them together.

Please note that neither the top object nor any of its direct or indirect successors is allowed to be selectable.

odb_ name	obj_ name	ex ist	offs x	offs y	offs z	rot x	rot y	rot z	ctor	mat	attrib	link
BAZ	o1		0.0	0.3	0.0	0.0	0.0	0.0	top			
	o1.o1		0.0	0.2	0.0	0.0	0.0	0.0	0.1 0.2 0.2 block			
	o1.o2		0.4	0.2	0.0	0.0	0.0	0.0	0.1 0.3 0.2 block			

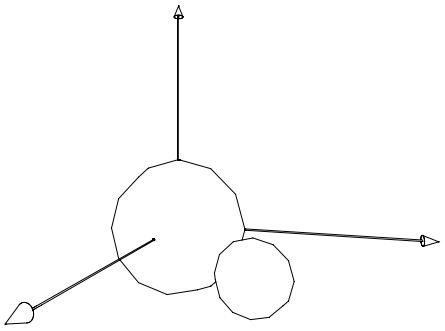


3.6.4 Sphere

The `sphere` function creates a homogeneous sphere beginning at the origin of the local coordinate system and extending to all sides.

The parameters are implemented by indicating a radius.

odb_ name	obj_ name	ex ist	offs x	offs y	offs z	rot x	rot y	rot z	ctor	mat	attrib	link
BAZ	o1		0.0	0.0	0.0	0.0	0.0	0.0	0.4 sphere			
	o2		0.3	0.0	0.5	0.0	0.0	0.0	0.2 sphere			



3.6.5 Hole

The `hole` function creates circular or rectangular holes in circular or rectangular areas. It can be used to simulate boolean operations (especially subtraction) in special cases. However, the actual subtraction in the sense of a boolean operation is not performed. The actual purpose of the `hole` function is to generate planes for the combinations circular outer line–rectangular hole, and rectangular outer line–circular hole. However, no outer planes along the outer line are created in the local Z-direction.

A hole object is created at the local origin, centered in relation to the outer plane. In the depth, a hole object starts at the origin of the local coordinate system and extends along the negative Z-axis.

As a general rule, a hole should always be contained in the contour. The hole should not touch the contour. Exception: the depths of the outline and the hole can be identical. In this case, the hole is transparent; in all other cases it has a base.

The creation parameters are as follows:

1. outline

   When the value `"R"` is indicated, the outer shape of the object is a rectangle, when the value `"C"` is indicated, it is a circle.

2. Outer width

   If the outer shape of the object is a rectangle, this value determines the outer width; in all other cases it indicates the radius of the contour.

3. Outer height

   If the outer shape of the object is a rectangle, this value determines the outer height; in all other cases the value is ignored.

4. Outer depth

   This value indicates the outer depth of the object.

5. Back plane

   This value controls the creation of a back plane in reference to the already generated front plane. If the value `1` is indicated, the plane will be created. In all other cases, indicate `0`.

6. Hole shape

   When the value `"R"` is indicated, the outer shape of the hole is a rectangle, when the value `"C"` is indicated, it is a circle.

7. Hole width

   If the outer shape of the hole is a rectangle, this value determines the hole's width; in all other cases it indicates the hole's radius.

8. Hole height

   If the outer shape of the hole is a rectangle, this value determines the hole's height; in all other cases the value is ignored.

9. Hole depth

   This value indicates the depth of the hole.

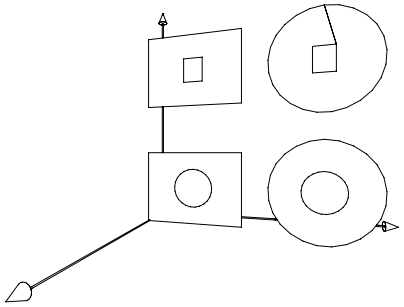
10. Hole offset in X-direction

    This value indicates the offset of the hole's center point in X-direction in reference to the local coordinate system's origin, which is also the center point of the object.

11. Hole offset in Y-direction

    This value indicates the offset of the hole's center point in Y-direction in reference to the local coordinate system's origin, which is also the center point of the object.

odb_ name	obj_ name	ex ist	offs x	offs y	offs z	rot	ctor	...
BAZ	o1		0.2	0.15	0.2	...	"R" 0.4 0.3 0.2 1 "C" 0.08 0.1 0.2 0.0 0.0 hole	
	o2		0.2	0.65	0.2	...	"R" 0.4 0.3 0.2 1 "R" 0.08 0.1 0.2 0.0 0.0 hole	
	o3		0.7	0.15	0.2	...	"C" 0.2 0.2 0.2 1 "C" 0.08 0.1 0.2 0.0 0.0 hole	
	o4		0.7	0.65	0.2	...	"C" 0.2 0.2 0.2 1 "R" 0.08 0.1 0.2 0.0 0.0 hole	



3.6.6 Parametric plane

The `surf` function creates a three-dimensional object based on a two-dimensional grid. The grid's coordinates function as support points that are connected by the resulting plane without edges. The specification for the `ctor` column is:

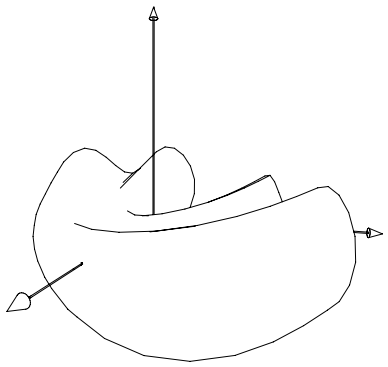
$$x_0\ y_0\ z_0\ \ldots x_{udim\times wdim-1}\ y_{udim\times wdim-1}\ z_{udim\times wdim-1}\ udim\ wdim\ umode\ wmode\ \texttt{surf}$$

The grid's dimensions are *udim* and *wdim*. Accordingly, for $udim \times wdim$ three-dimensional coordinates should be entered to define the grid. Within each basic plane, the right-hand-rule determines the orientation[15].

The *umode* and *wmode* flags can be used to indicate whether (`1`) or not (`0`) the plane should be closed along the corresponding grid direction.

The following example creates a parametric plane from 32 support points. The plane is closed along the grid direction and was subsequently turned.

```
-0.150815  0.064026 -0.919388  0.075575 -0.269460 -0.810568 \
 0.329356 -0.102473 -0.677988  0.576438 -0.273374 -0.555442 \
 0.772948  0.063867 -0.448708  0.563432  0.366948 -0.549524 \
 0.327520  0.171928 -0.673550  0.062688  0.355544 -0.804890 \
-0.490592 -0.095985 -0.379506 -0.371192 -0.281256 -0.307272 \
-0.237910 -0.188485 -0.220444 -0.107800 -0.283430 -0.139519 \
-0.004815 -0.096073 -0.070032 -0.115308  0.072304 -0.136939 \
-0.239162 -0.036040 -0.218088 -0.378620  0.065969 -0.304818 \
-0.565542 -0.069120  0.606108 -0.431332 -0.277990  0.529648 \
-0.275384 -0.175605  0.441702 -0.126945 -0.283430  0.357456 \
-0.004079 -0.074762  0.288498 -0.128341  0.115085  0.359282 \
-0.273712 -0.004849  0.441200 -0.432810  0.110994  0.531496 \
 0.122084 -0.013038  0.885150  0.226666 -0.273824  0.722364 \
 0.348838 -0.145497  0.535098  0.464742 -0.279946  0.355724 \
 0.561242 -0.018840  0.208880  0.464406  0.218190  0.359580 \
 0.350472  0.067949  0.534016  0.226248  0.212398  0.726268 \
8 4 1 0 surf
```



[15] With the thumb of the right hand vertical to the plane, the other fingers of that same hand indicate the orientation.

3.6.7 Polygon

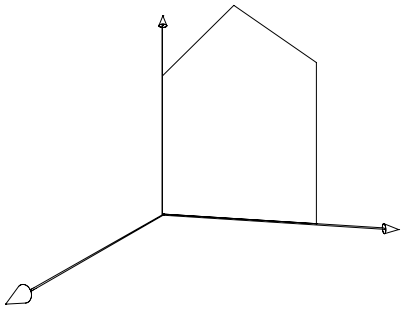
The `polyg` function creates a polygon in the space based on a list of three-dimensional coordinates. These should be indicated in clockwise direction. The last coordinate is automatically connected to the first. The described polygon should be simple, convex and planar. The generated polygon has no back plane.

The specification for the `ctor` column is:

$$x_0\ y_0\ z_0\ \ldots x_{n-1}\ y_{n-1}\ z_{n-1}\ n\ \texttt{polyg}$$

The following example creates a simple polygon:

```
0.0 0.0 0.0 0.0 0.7 0.0 0.35 1.0 0.0 0.7 0.7 0.0 0.7 0.0 0.0 5 polyg
```

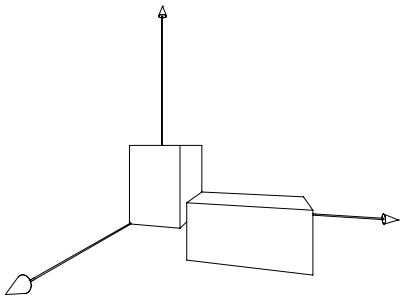


3.6.8 Block

The `block` function creates a homogeneous cube starting at the origin of the local coordinate system and extending along the local coordinate system's positive axes.

The parameters are implemented by indicating width, height, and depth.

odb_ name	obj_ name	ex ist	offs			rot			ctor	mat	attrib	link
			x	y	z	x	y	z				
BAZ	o1		0.0	0.0	0.0	0.0	0.0	0.0	0.2 0.3 0.4 block			
	o2		0.3	0.0	0.5	0.0	0.0	0.0	0.3 0.15 0.2 block			

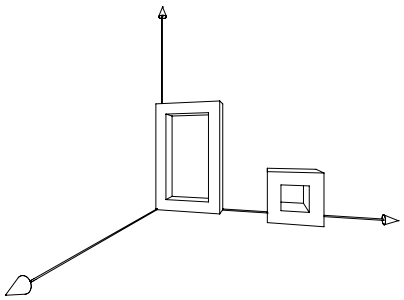


3.6.9 Frame

The `frame` function creates a frame starting at the origin of the local coordinate system and extending along the local coordinate system's positive axes. This is achieved by subtracting an orthogonal volume from the solid. The thickness of the frame is identical in X- and Y-direction. For the dimensions in X- and Y-direction w and h and for the x/y-thickness th, $w, h > 2 \times th$ should always apply.

The parameters are implemented by indicating frame width, height, depth, and thickness.

odb_ name	obj_ name	ex ist	offs x	offs y	offs z	rot x	rot y	rot z	ctor	mat	attrib	link
BAZ	o1		0.0	0.0	0.0	0.0	0.0	0.0	0.3 0.5 0.1 0.05 frame			
	o2		0.5	0.0	0.0	0.0	0.0	0.0	0.2 0.2 0.2 0.05 frame			



3.6.10 Rotating Solid Object

The `rot` function creates a three-dimensional object by rotating a three-dimensional definition curve. The specification for the `ctor` column is:

$$axis_x\ axis_y\ axis_z\ x_0\ y_0\ z_0\ \ldots x_{n-1}\ y_{n-1}\ z_{n-1}\ n\ angle\ smooth\ u\ w\ c0\ c1\ \texttt{rot}$$

The parameters $axis_x$, $axis_y$, and $axis_z$ specify the rotation axis using a standardized vector.

The parameters x_0 through z_{n-1} describe the definition curve, and the parameter n contains the number of the definition curve's coordinates. The definition curve and the rotation axis should always be in the same plane. Avoid coordinates that are located exactly on the rotation axis. The curve should be defined according to the right-hand-rule: When the thumb of the right hand points in the direction of the rotation, the other fingers on that hand show the orientation.

The *angle* parameter determines the angle. To create a homogeneous rotating solid body, enter the value 360.0.

The *smooth* parameter determines whether the coordinates of the definition curve are connected in a linear (`0`) way or using soft transitions(`1`).

The u indicates whether the last and first point of the definition curve should be connected (`1`) or not connected (`0`).

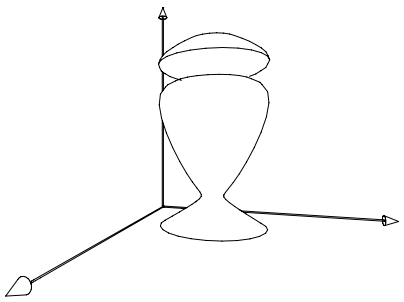
The w indicates, whether the solid is closed (`1`) along the rotation or not closed (`0`). If *angle* has the value 360.0, generally the value `1` is indicated, otherwise the value `0` is indicated.

The $c0$ parameter indicates whether for the first and last coordinate of the definition axis a linear lid plane, positioned vertically to the rotation axis, should (`1`) or should not (`0`) be created.

The $c1$ parameter indicates whether the inner planes being created should (`1`) or should not (`0`) be created if $angle < 360.0$ and w are not equal 1.

The following example shows how to create a homogeneous rotating solid body around a local Y-axis, which is subsequently moved in X- and Z-direction.

```
0.0 0.1 0.0 \
0.2 0.0 0.0 0.05 0.1 0.0 0.1 0.2 0.0 0.2 0.5 0.0 0.05 0.55 0.0 0.2 0.6 0.0 0.1 0.7 0.0 \
7 360.0 1 0 1 1 0 rot
```



The `rotx` function creates a rotating solid body around the local X-axis. The specification for the `ctor` column is:

$$x_0\ y_0\ \ldots x_{n-1}\ y_{n-1}\ n\ angle\ smooth\ u\ w\ c0\ c1\ \texttt{rotx}$$

The `roty` function creates a rotating solid body around the local X-axis. The specification for the `ctor` column is:

$$x_0\ y_0\ \ldots x_{n-1}\ y_{n-1}\ n\ angle\ smooth\ u\ w\ c0\ c1\ \texttt{roty}$$

The `rotz` function creates a rotating solid body around the local X-axis. The specification for the `ctor` column is:

$$y_0\ z_0\ \ldots y_{n-1}\ z_{n-1}\ n\ angle\ smooth\ u\ w\ c0\ c1\ \texttt{rotz}$$

3.6.11 Extrusion

The `sweep` function creates a three-dimensional object by dragging a three-dimensional curve in a predetermined direction. The specification for the `ctor` column is:

$$axis_x\ axis_y\ axis_z\ len\ x_0\ y_0\ z_0\ \ldots x_{n-1}\ y_{n-1}\ z_{n-1}\ n\ smooth\ u\ c0\ c1\ \texttt{sweep}$$

The parameters $axis_x$, $axis_y$ and $axis_z$ specify the dragging direction using a standardized vector.

The parameter *len* indicates the length along the dragging direction.

The parameters x_0 through z_{n-1} describe the definition curve, and the parameter n describes the number of the definition curve's coordinates. The definition curve should always be located in one plane to which the dragging vector should be vertical. The curve should be defined according to the right-hand-rule: When the thumb of the right hand points in the direction of the dragging vector, the other fingers on that hand show the orientation of the definition curve.

The *smooth* parameter determines whether the coordinates of the definition curve are connected in a linear (`0`) way or using soft transitions(`1`).

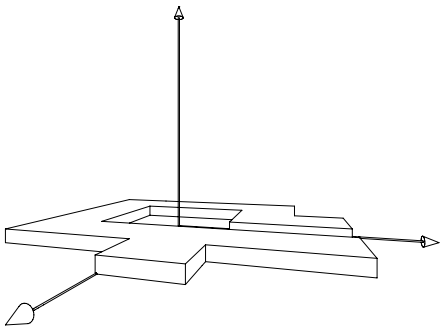
The u indicates whether, according to the value of *smooth*, the last and first point of the definition curve should be connected (`1`) or not connected (`0`) .

The $c0$ parameter indicates whether (`1`) or not (`0`) the solid should receive lid planes vertical to the dragging vector.

The $c1$ parameter indicates whether (`1`) or not (`0`) the connection between the last and firs coordinate should be handled as a plane. If both coordinates are in the same location, enter `0`.

The following example creates an extrusion along the local Y-axis.

```
0.0 1.0 0.0 0.05 \
0.5 0.0 -0.5 -0.5 0.0 -0.5 -0.5 0.0 0.5 0.0 0.0 0.5 0.0 0.0 0.7 0.25 0.0 0.7 \
0.25 0.0 0.5 0.7 0.0 0.5 0.7 0.0 0.25 -0.25 0.0 0.25 -0.25 0.0 -0.25 0.25 0.0 -0.25 \
0.25 0.0 0.1 0.7 0.0 0.1 0.7 0.0 -0.15 0.5 0.0 -0.15 16 0 0 1 1 sweep
```



The `sweepx` creates an extrusion along the local X-axis. The specification for the `ctor` column is:

$$len\ z_0\ y_0\ \ldots z_{n-1}\ y_{n-1}\ n\ smooth\ u\ c0\ c1\ \texttt{sweepx}$$

The `sweepy` creates an extrusion along the local Y-axis. The specification for the `ctor` column is:

$$len\ x_0\ z_0\ \ldots x_{n-1}\ z_{n-1}\ n\ smooth\ u\ c0\ c1\ \texttt{sweepy}$$

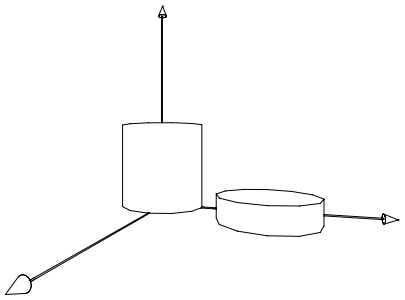
The `sweepz` creates an extrusion along the local Z-axis. The specification for the `ctor` column is:

$$len\ x_0\ y_0\ \ldots x_{n-1}\ y_{n-1}\ n\ smooth\ u\ c0\ c1\ \texttt{sweepz}$$

3.6.12 Cylinder

The `cyl` function creates a homogeneous cylinder symmetrical in rotation to the local Y-axis. The cylinder starts at the origin of the local coordinate system and extends along the coordinate system's positive Y-axis. The parameters are implemented by indicating two positive numbers for length and radius as arguments for the `cyl` function.

odb_	obj_	ex	offs			rot			ctor	mat	attrib	link
name	name	ist	x	y	z	x	y	z				
BAZ	o1		0.0	0.0	0.0	0.0	0.0	0.0	0.4 0.2 cyl			
	o2		0.5	0.0	0.2	0.0	0.0	0.0	0.1 0.2 cyl			



3.6.13 OFML Reference

The `clsref` function creates an instance of an OFML class. The specification for the `ctor` column is:

$$p_0 \ldots p_{n-1}\ n\ \texttt{"}\textit{classname}\texttt{"}\ \texttt{clsref}$$

where the class specific creation or initialization parameters are p_0 through p_{n-1}. The are mapped to the initialization function as follows:

$$\textit{classname}\texttt{::initialize(}\textit{pFa}\texttt{,}\ \textit{pNa}\texttt{,}\ p_0\texttt{,}\ \ldots\texttt{,}\ p_{n-1}\texttt{)}$$

classname is the optional, fully qualified name of the OFML class to be used. If the name is not or not fully qualified, the name of the package containing the ODB will automatically precede the name.

3.6.14 ODB Reference

The `odbref` function creates an instance of an ODB definition. The specification for the `ctor` column is:

$$p_0 \ldots p_{n-1}\ n\ \texttt{"}\textit{odbname}\texttt{"}\ \texttt{odbref}$$

where p_0 through p_{n-1} are the specific parameters you can access in the referenced ODB block using the ODB parameters `P0` through `P`*n-1*.

odbname is the optional, fully qualified name of the ODB definition to be used. If *odbname* is not or not fully qualified, the name of the package containing the ODB will automatically precede the name.

3.7 Material Assignment

The `mat` column is used for assigning materials. If it is not empty, it specifies a material or several materials as follows:

- If the entry references a primitive type, the material expression should return only one result as a material name. This material will be transferred to the object method `setMaterial()`.
- If the entry references a class type (`clsref`), the entry can contain any amount of material names, which can also be combined using vectors. They will be combined into a vector in the basic OFML language and transferred to the object method `setMaterials()` as an argument.

  Example. Let us assume that the `Mat` ODB parameter is set to `"foo"` and the material column contains the expression `"bar" $Mat "baz" 2]`. The method `setMaterials()` is invoked for the object as follows:

  ```
  obj.setMaterials(["bar", ["foo", "baz"]])
  ```

- If the entry references an ODB block (`odbref`), this entry can contain any amount of material names which can also be combined using vectors. Within the referenced ODB block, the materials can be accessed using the ODB parameters `M0` through `M`*(n-1)*.

Material names can optionally be indicated as fully qualified names. If they are not fully qualified or only partially qualified, the material name is automatically preceded by the package name containing the ODB.

3.8 Constructive Solid Geometry (CSG)

CSG allows the creation of complex shaped solid objects by combining primitive objects using Boolean operators. With the exception of the `stretch` operation, these operators are specified in column `ctor` via the function `csg`, their operands are the children in the object hierarchy.

The following regulations apply to these children:

1. Only elementary geometries (`ellipsoid`, `imp`, `sphere`, `surf`, `block`, `frame`, `rot`, `sweep` or `cyl`) and CSG nodes are allowed.
2. All geometries (especially those of type `imp` or `surf`) must be closed three-dimensional shapes.
3. Field `obj_name` only serves to define the hierarchy. Except for the topmost CSG node, no OFML objects are created.
4. All data in the fields `mat` and `attrib` are ignored. Therefore, these fields should be left blank.

The following subsections describe the available operators.

3.8.1 Union

The operation `union` generates the union (logical OR) of the geometries of their operands.

The following example shows a bar with a rounded end:

odb_ name	obj_ name	ex ist	offs x	offs y	offs z	rot x	rot y	rot z	ctor	mat	attrib	link
BAZ	o1		0.0	0.0	0.0	0.0	0.0	0.0	union csg			
	o1.o1		0.0	0.0	0.0	0.0	0.0	0.0	0.5 0.02 cyl			
	o1.o2		0.0	0.5	0.0	0.0	0.0	0.0	0.02 sphere			

3.8.2 Difference

The operation `diff` generates the difference of the geometries of their operands. In the case of more than two operands, operands 2..n first are united and then subtracted from the first operand.

The following example shows a block with a cylindrical hole:

odb_ name	obj_ name	ex ist	offs x	offs y	offs z	rot x	rot y	rot z	ctor	mat	attrib	link
BAZ	o1		0.0	0.0	0.0	0.0	0.0	0.0	diff csg			
	o1.o1		0.0	0.0	0.0	0.0	0.0	0.0	2.0 0.5 2.0 block			
	o1.o2		1.0	0.0	1.0	0.0	0.0	0.0	0.2 cyl			

3.8.3 Intersection

The operation `inter` generates the intersection (logical AND) of the geometries of their operands.

The following example shows a lens-like object:

odb_ name	obj_ name	ex ist	offs x	offs y	offs z	rot x	rot y	rot z	ctor	mat	attrib	link
BAZ	o1		0.0	0.0	0.0	0.0	0.0	0.0	inter csg			
	o1.o1		−0.8	0.0	0.0	0.0	0.0	0.0	1.0 sphere			
	o1.o2		0.8	0.0	0.0	0.0	0.0	0.0	1.0 sphere			

3.8.4 Stretch

The stretch operation is represented by the function *len a b c d* `stretch` in the `ctor` column. The parameters are defined as follows:

The parameter *len* specifies the lenght of the segment to insert, negative values are permitted and are interpreted as contraction.

The parameters *a b c d* describe the cutting plane in the form $ax + by + cz = d$. The vector *a b c* is the normal vector of the plane, *d* is the distance of the plane to the origin of the coordinate system.

The following example shows an object which is stretched by 0.5 units along the x-axis.

odb_ name	obj_ name	ex ist	offs x	offs y	offs z	rot x	rot y	rot z	ctor	mat	attrib	link
BAZ	o1		0.0	0.0	0.0	0.0	0.0	0.0	0.5 1.0 0.0 0.0 0.0 stretch			
	o1.o1		0.0	0.0	0.0	0.0	0.0	0.0	"sitz" 1.0 1.0 1.0 imp			

3.9 Attributes

The `attrib` column can contain zero or more of the following expressions.

3.9.1 Selectability

The object's selectability is explicitly prohibited through the expression `0 sel` and it is permitted through `1 sel`.

If there is no indication, and if the object was created by referencing an ODB block (using `odbref`) it is not selectable. If it was created by instancing an OFML class (using `clsref`), the object's selectability depends on the implementation of the OFML class. In all other cases, the object is a primitive object which is not selectable by default. The selectability for this object should not be permitted.

3.9.2 Collision Response

The expression `0 cd` excludes the object from the collision determination. If there is no entry or the expression `1 cd` is entered, the object is taken into consideration for the collision determination.

This expression should only be used for entries referencing an ODB block (using `odbref`) or an OFML class (using `clsref`).

If the collision determination is deactivated for an object, this deactivation also applies to its direct or indirect successors.

3.9.3 Editing Response

An object's response to editor operations such as using the clipboard (*clipboard*) is defined by an expression in the form of *value* `cut`. The following values are permissible for `value`:

`-1` In general, deleting the object is not permitted.

`0` Deleting the object itself is not permitted, however, it may be deleted through an upper level. In case of an attempted "cut"-operation (*cut, delete*) on the object, the operation is applied to the first object that can be cut in an upward traversing action.

`1` Deleting the object and copying it to the clipboard is permitted. This is applicable even when the editing response was not specified in an ODB entry.

`2` Deleting the object is permitted, however, it should not be copied to the clipboard.

3.9.4 Degree of Freedom for Translation

The `trx` function can be used to indicate whether or not the object can be moved in the direction of each axis on the local coordinate system. The function expects a single, integer argument resulting from an addition of the permitted axes, where the X-, Y-, and Z-axes are represented by 1, 2, and 4. If the argument is 0, the object cannot be moved.

3.9.5 Degree of Freedom for Rotation

Using the `rtx` function you can indicate for each axis on the local coordinate system whether or not the object can be rotated around the respective axis. The function expects a single, integer argument resulting from an addition of the permitted axes, where the X-, Y-, and Z-axes are represented by 1, 2, and 4. If the argument is 0, the object cannot be moved.

3.9.6 Properties

The `prop` function can be used to set optional parameters after object creation. This is done by calling the method `setPropValue()` (see interface `Property` in OFML specification[16]). The function expects two arguments: The first argument specifies the key of the property and must be an OFML symbol (i.e. including the leading `@` character). The second argument specifies the value to be set and must match the type of the property. This function may be called repeatedly to set an arbitrary number of properties.

3.9.7 Layer

With the function `layer` each object can be assigned to a layer (see section 6). The function expects a string argument containing the name of the layer.

3.10 Link

The `link` column is not supported in this version[17].

[16] EasternGraphics GmbH: *OFML – Standardisiertes Datenbeschreibungsformat der Büromöbelindustrie.*
[17] It will be included in later upgrades and links to other tables using a key.

4 Attachment Points

4.1 How To Use Attachment Points

When you add a new object to a plan and select an existing object, the system attempts to place the new object in relation to the selected object. For this purpose, the existing object needs an attachment point, and the object you are inserting needs a matching element[18].

Every attachment point can be identified by its own unique symbolic name. This name is used to assign matching attachment points.

The position of the attachment points is indicated in relation to the local coordinate system of the specific object. A new object is always placed in a way that its attachment point is at the same location with the existing object's matching attachment point. Furthermore, the new object can be rotated to a certain angle around the Y-axis intersecting this point.

In order to be able to add different object types in different locations, you can assign a list of attachment points for every object. When you are adding an object to an existing object, the system will browse the existing object's list from beginning to end until it finds an attachment point matching the one defined by the object you want to insert, and the new object can be placed without causing a collision

4.2 Defining Attachment Points

Table name: `attpt`
Obligatory table: yes

The attachment points for an object are described in the *attpt* table, which is described in the 8 table.

field-nummer	field name	description
1	`odb_name`	ODB–Name
2	`name`	symbolic name of attachment point
3	`select`	selection of attachment point
4	`text_idx`	index in text table
5	`x_pos`	local x–position of attachment point
6	`y_pos`	local y–position of attachment point
7	`z_pos`	local z–position of attachment point
8	`direction`	attachment direction
9	`rotation`	rotation of object to be inserted
10	`mode`	insert mode (child/neighbor)

Table 8: Definition of attachment points

In the following section the individual columns of this table are described in more detail:

[18] In addition to or instead of attachment points, the ODB planning element's class can also implement its own logic for attaching objects.

- `odb_name`
  Column `odb_name` contains the basic name of the ODB name for whom this attachment point definition is valid.

  When determining the attachment point of an object, the prefix of the ODB name determines the ODB in which the attachment points are to be searched for. The basic ODB name is used for the *attpt* table.

  In the current implementation, all entries of this table that correspond to the key are supplied as potential attachment points in the order of their appearance in the table, unless they have been explicitly deselected in column `select`.

- `name`
  Column `name` contains the symbolic name of the attachment point. It consists of any series of letters, digits, and underscores. The first character must not be a digit and the series is case-sensitive.

  To ensure that the names of the attachment points from different packages don't collide, the names of the attachment points should have a prefix that is as unique as possible and that e.g. may be comprised of the manufacturer or serial abbreviation of the packet. An exception to this rule is a combination of elements from different series **of a** manufacturer.

  The name of the attachment point should be unique within all entries of the *attpt* table with the same ODB name.

- `select`
  Column `select` is used for the selection of the attachment point. Using this column, an attachment point can be explicitly enabled or disabled. The attachment point is implicitly enabled,if this column is blank. If it is not blank, it must contain an expression in Reverse Polish Notation whose result is a numerical value. If the result is 0, the attachment point is disabled; otherwise, it is enabled.

- `text_idx`
  This column contains an index in a text table containing an attachment point describing the text. The text can be used in a tree consisting of an object and attachment point hierarchy in the user interface[19].

- `x_pos`, `y_pos`, `z_pos`
  These columns contain the local coordinates of the position of the attachment point in the form of an expression in Reverse Polish Notation.

- `direction`
  In column `direction`, the direction is set in which this attachment point should be inserted. The predefined directions are as follows: `R` (right), `L` (left), `B` (back), `F` (front), and `T` (top). In addition, any other directions can be defined.

  If the column is blank, there is no concrete preset attachment direction. This is relevant for determining the opposite attachment points using the table *oppattpt* that is described in section 4.3, since all attachment points that are contained in this table and are identical by name are considered opposite attachment points regardless of a direction that may have been specified.

[19] In the current implementation, the column is not used and should contain 0.

- `rotation`
  In column `rotation`, you can specify a rotation of the object to be inserted around the y axis that passes through the attachment point. You specify this mathematically positive (counterclockwise) as an expression in Reverse Polish Notation using degree measure.

- `mode`
  In column `mode` you determine if the object to be inserted is to be inserted as a child or neighbor of the existing object. To insert it as a child, this column must contain a `C`; otherwise it must contain a `S`.

4.3 Definition of opposite attachment points

Table name: `oppattpt`
Obligatory table: yes

The table *oppattpt* described in table 9 determines which attachment points from different objects match each other. This is determined from the point of view of the object to be inserted which supplies a list of its own matching attachment points for possible attachment points of other objects taking their attachment direction into account[20].

field number	field name	description
1	`odb_name`	ODB name
2	`select`	selection of opposite attachment point
3	`opposite`	name of opposite attachment point
4	`direction`	direction of opposite attachment point
5	`att_points`	list of its own matching attachment points

Table 9: opposite attachment points

The following section describes the columns listed in table 9 in greater detail:

- `odb_name`
  Column `odb_name` contains the basic name of the ODB name of the object to be inserted. The attachment points of the object are listed in column `att_points`.

- `select`
  Column `select` is used to select the opposite attachment point specified in column `opposite`. You can explicitly enable or disable the attachment point in this column. The attachment point is implicitly enabled if the column is blank. If it is not blank, it must contain an expression in Reverse Polish Notation whose result is a numerical value. If the result is 0, the opposite attachment point is disabled; otherwise, it is enabled.

- `opposite`
  Column `opposite` contains the name of the opposite attachment point. In addition to

[20]The object to be inserted is requested to supply a list of its own attachment points that may be possible counterparts of the currently viewed attachment point of the existing object.

the ODB name in column `odb_name` and the direction in column `direction`, is used as a key when accessing table *oppattpt*.

- `direction`
  Column `direction` contains the direction of the opposite attachment point. In addition to the ODB name in column `odb_name` and the name of the opposite attachment point in column `opposite`, it is used as a key when accessing table *oppattpt*.

  The opposite attachment point is only considered either if no attachment direction was specified for it, or if the `direction` field in this table is blank, of if the specified attachment direction is identical to the direction indicated in this table.

- `att_points`
  Column `att_points` contains a list of attachment points of the objects to be inserted with the ODB names indicated in column `odb_name`. The names match the opposite attachment point specified in column `opposite`. The list is delimited by blank characters.

4.4 Standard attachment points

Table name: `stdattpt`
Obligatory table: yes

In addition to the user-defined attachment points, there is a set of 18 standard attachment points, that are located in the eight corners, in the center of the top and bottom edges, and in the middle of the deck and floor areas of the terminating volume of an OFML object. The order and attachment direction of these attachment points are dependent on the current planning direction. The names of these standard attachment points are described in table 10.

name		position
bottom	top	
LBF	LTF	left front corner
CBF	CTF	middle of the front edge
RBF	RTF	right front corner
LBC	LTC	middle of the left edge
CBC	CTC	middle of the floor or deck area
RBC	RTC	middle of the right edge
LBB	LTB	left back corner
CBB	CTB	middle of the back edge
RBB	RTB	right back corner

Table 10: names of standard attachment points

The first letter of the name of a standard attachment point determines the position of the attachment point in x direction (left: `L`, middle: `C`, right: `R`). The second letter determines its position in y direction (bottom: `B`, top: `T`). Finally, the third letter determines its position in z direction (front: `F`, middle: `C`, back: `B`).

Using table *stdattpt* described in table 11, it is possible to control the use of standard attachment points for ODB objects by their ODB names. In particular, you can determine if

standard attachment points should be used at all for an object with a given ODB name, and if so, whether these are to be taken into account before or after the user-defined attachment points. Further, it is possible to only take a subset of the standard attachment points into account.

field number	field name	description
1	`odb_name`	ODB name
2	`has_stdattpts`	general use of standard attachment points
3	`prep_stdattpts`	position of standard attachment points
4	`stdattpts`	selection of subset of standard attachment points

Table 11: standard attachment points

The following section describes the individual columns in table *stdattpt* in greater detail.

- `odb_name`
  Column `odb_name` contains the basic name of the ODB name of the respective object.
- `has_stdattpts`
  Column `has_stdattpts` determines if objects with the appropriate ODB name have standard attachment points or not. The column must contain an unsigned integer value. If this value is 0, standard attachment points are not used regardless of the content of the other columns in this table. Otherwise, the standard attachment points are used as indicated in the following columns.
- `prep_stdattpts`
  Column `prep_stdattpts` determines, if the standard attachment points are to be viewed before or after any user-defined attachment points. It must contain an unsigned integer value. If this value is 0, they are viewed after the user-defined points[21], or before them.
- `stdattpts`
  Column `stdattpts` either is blank or contains a list of standard attachment point names that are delimited by blank characters. In the first case, all standard attachment points are taken into account; in the second case, only the specified ones are.

If there is no entry in the *stdattpt* table for a ODB name, all standard attachment points after any user-defined attachment points are taken into account for objects with this ODB name.

[21] This is probably the normal case.

5 Functions

Table name: `funcs`
Obligatory table: no

Functions can have two distinguishing characteristics: built-in and user-defined functions.

In principle, the function arguments are noted before invoking the function. For example, in order to calculate the square root of 2.0, one must write "'`2.0 sqrt`"'.

Generally it must be said that the possibilities offered by ODB for processing and defining functions are hardly used to their fullest extent, at least for generating 2D geometries. Normally, used expressions and functions used by 2D ODB are limited to processing arithmetic standard operators `+`, `-`, `*`, and `/`.

5.1 Built-in Functions

In addition to the functions regarding object generation and setting of attributes described in sections 2 and 3, particularly mathematical functions are built into the interpreter for the expressions used in ODB.

The return value of some of theses functions is a frequently used constant. For example, the `M_PI` function returns the value of π. For other functions, the return value depends on one or several arguments that the function expects. One example is the function `sin` that calculates the sine of its argument using the radian measure.

Table 12 contains a summary of all built-in functions that return constants. The built-in mathematical functions are listed in table 13. Table 14 describes the built-in function for manipulating the stack. Table 15 documents the use of two functions that are particularly interesting for 2D ODB.

Name	returned value	name	returned value
`M_1_PI`	$1/\pi$	`M_2_PI`	$2/\pi$
`M_2_SQRTPI`	$2/\sqrt{\pi}$	`M_2PI`	2π
`M_E`	e	`M_LN10`	$\ln 10 = \log_e 10$
`M_LN2`	$\ln 2 = \log_e 2$	`M_LOG10E`	$\lg e = \log_{10} e$
`M_LOG2E`	$1/\ln 2 = \log_2 e$	`M_PI`	π
`M_PI_2`	$\pi/2$	`M_PI_4`	$\pi/4$
`M_SQRT1_2`	$1/\sqrt{2}$	`M_SQRT2`	$\sqrt{2}$

Table 12: built-in constants

5.2 User-defined Functions

User-defined functions are placed into the function table. The structure of this function table is shown in table 16.

Arguments	name	event	description
x	`acos`	y	$y = \arccos x$
x	`asin`	y	$y = \arcsin x$
x	`atan`	y	$y = \arctan x$
x y	`atan2`	z	$z = \arctan(y/x)$ The signs of x and y are used to calculate the quadrant of the event.
x	`ceil`	y	Calculates the smallest integer value y that is greater than or equal to x.
x	`cos`	y	$y = \cos x$
x	`cosh`	y	$y = \cosh x$
x	`exp`	y	$y = e^x$
x	`fabs`	y	$y = \|x\|$
x	`floor`	y	Calculates the largest integer value y that is smaller than or equal to x.
x y	`fmod`	z	Calculates the floating point remainder of x/y.
x	`log`	y	$y = \ln x$
x	`log10`	y	$y = \lg 10$
x	`modf`	i f	Divides the x argument into the integer part i and the fractional part f so both have the same sign as x.
x	`neg`	y	$y = -x$
x y	`pow`	z	$z = x^y$
x	`sin`	y	$y = \sin x$
x	`sinh`	y	$y = \sinh x$
x	`sqrt`	y	$y = \sqrt{x}$
x	`tan`	y	$y = \tan x$
x	`tanh`	y	$y = \tanh x$

Table 13: built-in mathematical functions

The first column contains the name of the function. The function name can consist of a series of letters of any length[22], digits, and underscores. The first character must be a letter or an underscore[23].

The second column contains the body of the function in the form of an expression in Reverse Polish Notation.

5.2.1 Function Arguments

A user-defined function can have any number of arguments, including none.

No special measures have to be taken for functions without argument.

For functions with arguments, the number of arguments must be at the beginning of the function body, followed by the invocation of the built-in `argc` function. This makes it possible to remove the specified number of arguments from the local stack of the expression

[22]Only letters `A` to `Z` and `a` to `z` are permitted. Therefore, an umlaut cannot be used.

[23]We recommend against using an underscore at the beginning of a function name, since such names are reserved for internal use.

Arguments	name	event	description
n	`argc`		The `argc` function must be invoked directly at the beginning of a user-defined function. The n parameter is the number of arguments that this user-defined function expects. It removes this number of values from the stack of the invoker and makes it available for the argument access using $\$a$.
x	`dup`	$x\ x$	The `dup` function duplicates the top element on the stack.
$x\ y$	`dup2`	$x\ y\ x$	The `dup2` function duplicates the element on the stack that is second to the top.
$s_i \ldots s_2\ s_1\ x$	`dupx`	$s_i \ldots s_2\ s_1\ s_x$	The `dupx` function duplicates the xth object from the top of the stack.
x	`pop`		The `pop` function removes the top element from the stack.
$x\ y$	`swap`	$y\ x$	The `swap` function swaps the two top elements on the stack.
$s_x \ldots s_2\ s_1\ x$	`swapx`	$s_1 \ldots s_2\ s_x$	The `swapx` function swaps the top element of the stack with the xth element from the top of the stack.

Table 14: functions for stack manipulation

Arguments	name	event	description
x	`utos`	s	The `utos` function changes the x floating point value to the s string according to the settings for unit formatting in the user interface.
x	`atos`	s	The `atos` function changes the a floating point value to the s string according to the settings for angle formatting in the user interface .

Table 15: functions for 2D ODB

Field number	field name	description
1	`name`	function name
2	`body`	function body

Table 16: function table

that invoked the function and to temporarily store them for the duration of processing the function. Then, the function can access the arguments using $\$n$, with n being the number of the argument. Numbering of the arguments begins at 0.

5.2.2 Return Value

A user-defined function can have any number of return values, including none.

To return one or several values, the values are simply left on the stack once the processing of the function bodies is completed. After the return of the function, these values are listed in the same order at the top of the local stack of the expression that invoked the function. It is the responsibility of the invoking expression to remove the return values from the stack.

5.2.3 Example

The following example shows the `DIST` function that calculates the distance between two points that are defined by their x and y coordinates.The arguments are put on the stack by the invoker in the following order: x_0, y_0, x_1, y_1. If the function returns, the arguments have been removed from the stack and are replaced by the result of the function.

`name`	`body`
`DIST`	`4 argc $2 $0 - dup * $3 $1 - dup * + sqrt`

The following table shows the processing of the function body using the local stack of the function.

Stack	token	operation
	`4`	`4` ⇒ Stack
`4`	`argc`	Applying the 4 argument values from the stack of the invoking expression
	`$2`	x_1 ⇒ Stack
x_1	`$0`	x_0 ⇒ Stack
x_1 x_0	`-`	$dx = x_1 - x_0$; x_1 and x_0 remove from stack and replace with dx
dx	`dup`	dx ⇒ Stack
dx dx	`*`	$dx^2 = dx \times dx$; remove both dx from stack and replace with dx^2
dx^2	`$3`	y_1 ⇒ Stack
dx^2 y_1	`$1`	y_0 ⇒ Stack
dx^2 y_1 y_0	`-`	$dy = y_1 - y_0$; remove y_1 and y_0 from stack and replace with dy
dx^2 dy	`dup`	dy ⇒ Stack
dx^2 dy dy	`*`	$dy^2 = dy \times dy$; remove both dy from stack and replace with dy^2
dx^2 dy^2	`+`	$sq = dx^2 + dy^2$; remove both dx^2 and dy^2 from stack and replace with sq
sq	`sqrt`	$dist = \sqrt{sq}$; remove sq from stack and replace with $dist$
$dist$		return value

6 Layers

Table name: `layer`
Obligatory table: no

6.1 Functioning of Layers

By means of the respective `layer`–function (see sections 2.8.7 resp. 3.9.7) objects can be assigned to an layer. This allows to assign properties such as visibility, color, etc. simultaneously to multiple objects, regardless of their position in the object hierarchy.

6.2 Definition of Layers

The properties of 2D layers are exclusively defined by the application.

3D layers are defined via table *layer*. The definition of layers is optional. For non-defined layers preset values are used. The values in this table on their part are default values that can be overwritten by the application.

field-number	field-name	description
1	`layer_name`	name of the layer
2	`attributes`	properties

Table 17: Definition of 3D Layers

In the following the individual fields of this table are described in more detail:

- `layer_name`
  This field specifies the name of the layer. The following characters can be used: all alphanumeric characters, `_` (underscore), `-` (hyphen) and `$` (dollar sign).

  Layer names should conform to the OLAYERS specification[24].

- `attributes`
  The layer properties are defined in this field by means of predefined functions. The function calls are formulated in Reverse Polish Notation, i.e., the arguments are stated in front of the function name.

  Currently only the function `visible` is defined. If the argument of this function has the integer value 0, then the objects on the layer are invisible. This affects rendering (real-time, photo-realism), printing and graphics export.

[24] Verband Büro-, Sitz- und Objektmöbel e.V.: *OLAYERS – OFML compatible Layers.*