

OpenSCAD Help

Table of contents

Introduction	11
What's new	12
Getting Started	12
Language Reference	13
Introduction	14
Comments	15
Values and Data Types	15
Numbers	15
Boolean Values	16
Strings	16
Ranges	17
The Undefined Value	17
Variables	18
Undefined variable	18
Scope of variables	19
Variables are set at compile-time, not at run-time	19
Special Variables	20
Vectors	20
Vector Operators (concat & len)	21
Matrix	22
Getting input	22
3D Objects	23
Primitive Solids	23
cube	23
sphere	24
cylinder	25
polyhedron	28
Debugging polyhedra	31
Mis-ordered faces	32
Point repetitions in a polyhedron point list	33
3D to 2D Projection	35
2D Objects	37
square	37
circle	38
ellipse	39
regular polygon	39
polygon	40
Text	43
Using Fonts & Styles	44
Vertical Alignment	46
Horizontal Aligment	46
3D Text	47
2D to 3D Projection	47
Linear Extrude	48
Usage	49
Twist	49
Center	51

Mesh Refinement	52
Scale	53
Rotate Extrude	54
Usage	55
Examples	55
Mesh Refinement	56
Extruding a Polygon	58
Description of extrude Parameters	60
Transforms	61
scale	61
resize	62
rotate	63
Rotation rule Help	65
translate	66
mirror	67
multimatrix	68
color	70
offset	72
minkowski	74
hull	75
Combining transformations	76
Boolean Combinations	76
union	79
difference	79
intersection	81
render	81
Condition and Iterator functions	82
For loop	82
Intersection For loop	85
if statement	87
else if	88
Conditional ?	88
Recursive function calls	88
Let Statement	89
Mathematical Operators	89
Scalar Arithmetical Operators	89
Relational Operators	89
Logical Operators	90
Conditional Operator	90
Vector-Number Operator	91
Vector Operators	91
Vector Dot-Product Operator	91
Matrix Multiplication	92
Trigonometric Functions	92
cos	92
sin	93
tan	93
acos	94
asin	94
atan	94

atan2	94
Other Mathematical Functions	94
abs	94
ceil	95
concat	95
cross	95
exp	95
floor	96
ln	96
len	96
let	97
log	97
lookup	97
max	98
min	98
norm	99
pow	99
rands	100
round	101
sign	101
sqrt	101
Infinities and NaNs	101
String Functions	102
str	102
chr	102
ord	103
Also see search()	103
List Comprehensions	103
Basic Syntax	103
Multiple generator expressions	104
for	104
each	106
if	106
if/else	107
let	107
Nested loops	107
Advanced Examples	108
Generating vertices for a polygon	108
Flattening a nested vector	109
Sorting a vector	109
Selecting elements of a vector	110
Concatenating two vectors	110
Other Language Features	110
Special Variables	110
\$fa, \$fs and \$fn	111
\$t	112
\$vp, \$vpt and \$vpd	113
\$preview	114
Echo Statements	114
Echo Function	115

Render	116
Surface	116
Search	119
Search Usage	119
Search Arguments	119
Search Usage Examples	120
OpenSCAD Version	122
parent_module(n) and \$parent_modules	122
assert	123
User Defined Functions and Modules	124
Introduction	124
Scope	124
Functions	124
Recursive Functions	124
Function Literals	124
Modules	124
Object modules	125
Operator Modules	125
Children	125
Further Module Examples	125
Recursive Modules	125
Overwriting built-in modules	125
Overwriting built functions	125
Debugging Aids	125
Advanced Concept	125
Background Modifier	125
Debug Modifier	125
Root Modifier	125
Disable Modifier	125
Echo Statements	126
External Libraries and code files	126
Use and Include	126
Directory separators	126
Variables	126
Scope of Variables	126
Overwriting variables	126
Example "Ring-Library"	126
Nested Include and Use	126
Import	126
Parameters	126
Convexity	126
Notes	127
Import DXF	127
Import STL	127
surface	127
Parameters	127
Text file Format	127
Images	127
Examples	127
MCAD Library	127

regular_shapes.scad	129
2D regular shapes	129
regular_polygon(sides, radius)	129
n-gons 2D shapes	129
triangle(radius)	129
pentagon(radius)	129
hexagon(radius)	129
heptagon(radius)	129
octagon(radius)	129
nonagon(radius)	130
decagon(radius)	130
hendecagon(radius)	130
dodecagon(radius)	130
ring(inside_diameter, thickness)	130
ellipse(width, height)	130
egg_outline(width, thickness)	130
3D regular shapes	130
cone	130
oval_prism	130
oval_tube	130
cylinder_tube	130
triangle_prism	130
triangle_tube	131
pentagon_prism	131
pentagon_tube	131
hexagon_prism	131
hexagon_tube	131
heptagon_prism	131
heptagon_tube	131
octagon_prism	131
octagon_tube	131
nonagon_prism	131
decagon_prism	131
hendecagon_prism	131
dodecagon_prism	132
torus	132
torus2	132
oval_torus	132
triangle_pyramid	132
square_pyramid	132
egg	132
involute_gears.scad	132
bevel_gear_pair()	132
bevel_gear()	132
gear()	132
Tests	132
test_gears()	132
test_meshing_double_helix()	133
test_bevel_gear()	133
test_bevel_gear_pair()	133

test_backlash()	133
dotSCAD Library	133
2D modules	133
arc	133
pie	133
rounded_square	133
line2d	133
polyline2d	133
hull_polyline2d	133
hexagons	134
polytransversals	134
multi_line_text	134
voronoi2d	134
3D modules	134
rounded_cube	134
rounded_cylinder	134
crystal_ball	134
line3d	134
polyline3d	134
hull_polyline3d	134
function_grapher	134
sweep	134
loft	135
starburst	135
voronoi3d	135
Transformations	135
along_width	135
hollow_out	135
bend	135
shear	135
2D functions	135
in_shape	135
bijection_offset	135
trim_shape	135
triangulate	136
contours	136
2D/3D functions	136
cross_sections	136
paths2sections	136
path_scaling_sections	136
bezier_surface	136
bezier_smooth	136
midpt_smooth	136
in_polyline	136
Paths	136
arc_path	136
bspline_curve	136
bezier_curve	137
helix	137
golden_spiral	137

archimedean_spiral	137
sphere_spiral	137
torus_knot	137
Extrusion	137
box_extrude	137
ellipse_extrude	137
stereographic_extrude	137
rounded_extrude	137
bend_extrude	137
2D Shape	138
shape_taiwan	138
shape_arc	138
shape_pie	138
shape_circle	138
shape_ellipse	138
shape_square	138
shape_trapezium	138
shape_cyclicpolygon	138
shape_pentagram	138
shape_starburst	138
shape_superformula	138
shape_glue2circles	138
shape_path_extend	139
2D Shape extrusions	139
path_extrude	139
ring_extrude	139
helix_extrude	139
golden_spiral_extrude	139
archimedean_spiral_extrude	139
sphere_spiral_extrude	139
Utils	139
util/sub_str	139
util/split_str	139
util/parse_number	139
util/reverse	140
util/slice	140
util/sort	140
util/rand	140
util/fibseq	140
util/bsearch	140
util/has	140
util/dedup	140
util/flat	140
Matrix	140
matrix/m_cumulate	140
matrix/m_translation	140
matrix/m_rotation	140
matrix/m_scaling	141
matrix/m_mirror	141
matrix/m_shearing	141

Point Transformation	141
ptf/ptf_rotate	141
ptf/ptf_x_twist	141
ptf/ptf_y_twist	141
ptf/ptf_circle	141
ptf/ptf_bend	141
ptf/ptf_ring	141
ptf/ptf_sphere	141
ptf/ptf_torus	141
Turtle	142
turtle/turtle2d	142
turtle/turtle3d	142
turtle/t2d	142
turtle/t3d	142
Pixel	142
pixel/px_line	142
pixel/px_polyline	142
pixel/px_circle	142
pixel/px_cylinder	142
pixel/px_sphere	142
pixel/px_polygon	142
pixel/px_from	142
pixel/px_ascii	143
pixel/px_gray	143
Part	143
part/connector_jpg	143
part/cone	143
part/join_T	143
Surface	143
surface/sf_square	143
surface/sf_bend	143
surface/sf_ring	143
surface/sf_sphere	143
surface/sf_torus	143
surface/sf_solidity	144
Noise	144
noise/nz_perlin1	144
noise/nz_perlin1s	144
noise/nz_perlin2	144
noise/nz_perlin2s	144
noise/nz_perlin3	144
noise/nz_perlin3s	144
noise/nz_worley2	144
noise/nz_worley2s	144
noise/nz_worley3	144
noise/nz_worley3s	144
noise/nz_cell	144
BOSL Library	145
Commonly Used	145
transforms.scad	145

Translations	145
move()	145
xmove()	147
ymove()	148
zmove()	148
left()	148
right()	148
fwd() / forward()	148
back()	148
down()	148
up()	148
shapes.scad	148
masks.scad	148
threading.scad	148
paths.scad	148
beziers.scad	149
Standard Parts	149
involute_gears.scad	149
joiners.scad	149
sliders.scad	149
metric_screws.scad	149
linear_bearings.scad	149
nema_steppers.scad	149
phillips_drive.scad	149
torx_drive.scad	149
wiring.scad	149
Miscellaneous	149
constants.scad	149
math.scad	150
convex_hull.scad	150
quaternions.scad	150
triangulation.scad	150
debug.scad	150
Nut Job Library	150

Introduction

OpenSCADHelp

OpenSCAD Help Project is a compilation of help information about OpenSCAD and some libraries.

Any programming language has two parts to learn. The language syntax and himself instructions and the common libraries of constructed instructions.

OpenSCAD Help is a compilation all this help in a unique site, with a unique form of search and found help about almost the most important things of OpenSCAD.

The libraries included in this project are the next:

MCAD

Components commonly used in designing and mocking up mechanical designs

<https://github.com/openscad/MCAD>

dotSCAD

Reduce the burden of 3D modeling in mathematics. <https://github.com/JustinSDK/dotSCAD>

BOSL

The Belfry OpenScad Library - A library of tools, shapes, and helpers to make OpenSCAD easier to use. <https://github.com/revarbat/BOSL>

Syntax

```
var = value;
module name(_) { _ }
name();
function name(_) = _
name();
include <...scad>
use <...scad>
```

2D

```
circle(radius)
square(size,center)
square([width,height],center)
polygon([points])
polygon([points],[paths])
```

3D

```
sphere(radius)
cube(size)
cube([width,height,depth])
cylinder(h,r,center)
cylinder(h,r1,r2,center)
polyhedron(points, triangles, convexity)
```

Transformations

```
translate([x,y,z])
rotate([x,y,z])
scale([x,y,z])
mirror([x,y,z])
multmatrix(m)
color("colorname")
color([r, g, b, a])
hull()
minkowski()
```

Boolean operations

```
union()
difference()
intersection()
```

Modifier Characters

```
*  disable
!  show only
#  highlight
%  transparent
```

Mathematical

```
abs
sign
acos
asin
atan
atan2
sin
cos
floor
round
ceil
ln
len
log
lookup
min
max
pow
sqrt
exp
rands
```

Other

```
echo(_)
str(_)
for (i = [start:end]) { _ }
for (i = [start:step:end]) { _ }
for (i = [...]) { _ }
intersection_for(i = [start:end]) { _ }
intersection_for(i = [start:step:end]) { _ }
intersection_for(i = [...]) { _ }
if (..) { _ }
assign (..) { _ }
search(..)
import("...stl")
linear_extrude(height,center,convexity,twist,slices)
rotate_extrude(convexity)
surface(file = "...dat",center,convexity)
projection(cut)
render(convexity)
```

Special variables

```
$fa minimum angle
$fs minimum size
$fn number of fragments
$t animation step
```

Created with the Personal Edition of HelpNDoc: [Easily create Qt Help files](#)

What's new

2020 April 22th.

- First version of PDF help file.
- First version of HTML help site.
- First version of EPUB help file.

Created with the Personal Edition of HelpNDoc: [Easy CHM and documentation editor](#)

Getting Started

OpenSCAD is a software for creating solid 3D CAD objects.

It is **free software** and available for **GNU/Linux**, Microsoft Windows and Mac OS X.

OpenSCAD focuses on the **CAD** aspects as opposition to artistic aspects. So it might be the application you are looking for when you are planning to create 3D models of machine parts.

OpenSCAD is not an interactive modeler. It is something like a 2D/3D-compiler that reads in a program file that describes the object and renders the model from this file. This gives you (the designer) full control over the modeling process. This enables you to easily change any step in the modeling process and make designs that are defined by configurable parameters.

OpenSCAD has two main operating modes:

- Preview is relatively fast using **3D graphics** and the computer's **GPU**, but is an approximation of the model and can produce **artifacts**; Preview uses **OpenCSG** and **OpenGL**.
- Render generates exact geometry and a fully **tessellated mesh**. It is not an approximation and as such it is often a lengthy process, taking minutes or hours for larger designs. Render uses **CGAL** as its geometry engine.

OpenSCAD provides two types of 3D modelling:

- **Constructive Solid Geometry (CSG)**
- extrusion of 2D primitives into 3D space.

OpenSCAD can be downloaded from <https://www.openscad.org/>. **OpenSCAD** is a software for creating solid 3D CAD objects.

It is **free software** and available for **GNU/Linux**, Microsoft Windows and Mac OS X.

Unlike most free software for creating 3D models (such as the well-known application **Blender**), OpenSCAD does not focus on the artistic aspects of 3D modelling, but instead focuses on the **CAD** aspects. So it might be the application you are looking for when you are planning to create 3D models of machine parts, but probably is not what you are looking for when you are more interested in creating computer-animated movies or organic life-like models.

OpenSCAD, unlike many CAD products, is not an interactive modeler. Instead it is something like a 2D/3D-compiler that reads in a program file that describes the object and renders the model from this file. This gives you (the designer) full control over the modelling process. This enables you to easily change any step in the modelling process and make designs that are defined by configurable parameters.

OpenSCAD has two main operating modes, *Preview* and *Render*. Preview is relatively fast using **3D graphics** and the **computer's GPU**, but is an approximation of the model and can produce **artifacts**; Preview uses **OpenCSG** and **OpenGL**. Render generates exact geometry and a fully **tessellated mesh**. It is not an approximation and as such it is often a lengthy process, taking minutes or hours for larger designs. Render uses **CGAL** as its geometry engine.

OpenSCAD provides two types of 3D modelling:

- **Constructive Solid Geometry (CSG)**
- extrusion of 2D primitives into 3D space.

Autocad DXF files are used as the data exchange format for 2D outlines. In addition to 2D paths for extrusion it is also possible to read design parameters from DXF files. Besides DXF files, OpenSCAD can read and create 3D models in the **STL** and **OFF** file formats.

OpenSCAD can be downloaded from <https://www.openscad.org/>. More information is available on the **mailing list**. OpenSCAD can also be tried online at <http://openscad.net/>, which is a partial port of OpenSCAD for the web. More information is available on the **mailing list**. OpenSCAD can also be tried online at <http://openscad.net/>, which is a partial port of OpenSCAD for the web.

Created with the Personal Edition of HelpNDoc: [Full-featured multi-format Help generator](#)

Language Reference

Table of Contents

1. [The OpenSCAD Language - General](#)
2. [3D Objects, Projection](#)
3. [2D Objects, Primitives, Text, Extrusion to 3D](#)
4. [Transformations](#)
5. [Boolean operations](#)
6. [Conditional and iterator functions](#)
7. [Mathematical operators](#)
8. [Mathematical functions](#)
9. [String functions](#)
10. **Type test functions**
11. [List comprehensions](#)

12. [Other language features](#)
13. [User defined functions and modules](#)
14. [Debugging aids - modifier characters](#)
15. **Importing geometry, Exporting geometry**

Created with the Personal Edition of HelpNDoc: [Full-featured EBook editor](#)

Introduction

OpenSCAD is a 2D/3D and solid modeling program which is based on a Functional programming language used to create models that are previewed on the screen, and rendered into 3D mesh which allows the model to be exported in a variety of 2D/3D file formats.

A script in the OpenSCAD language is used to create 2D or 3D models. This script is a free format list of action statements.

```
object();
variable = value;
operator() action();
operator() {action(); action();}
operator() operator() {action(); action();}
operator() {operator() action();
               operator() {action(); action();}}
```

Objects

Objects are the building blocks for models, created by 2D and 3D primitives. Objects end in a semicolon ';'.

Actions

Action statements include creating objects using primitives and assigning values to variables. Action statements also end in a semicolon ';'.

Operators

Operators, or transformations, modify the location, color and other properties of objects. Operators use braces '{}' when their scope covers more than one action. More than one operator may be used for the same action or group of actions. Multiple operators are processed Right to Left, that is, the operator closest to the action is processed first. Operators do not end in semicolons ';', but the individual actions they contain do.

Examples

```
cube(5);
x = 4 + y;
rotate(40) square(5,10);
translate([10,5]) { circle(5); square(4); }
rotate(60) color("red") { circle(5); square(4); }
color("blue") { translate([5,3,0]) sphere(5); rotate([45,0,45]) { cylinder(10); cube([5,6,7]); } }
```

Comments

Comments are a way of leaving notes within the script, or code, (either to yourself or to future programmers) describing how the code works, or what it does. Comments are not evaluated by the compiler, and should not be used to describe self-evident code.

OpenSCAD uses C++-style comments:

```
// This is a comment

myvar = 10; // The rest of the line is a comment

/*
Multi-line comments
can span multiple lines.
*/
```

Values and Data Types

A value in OpenSCAD is either a Number (like 42), a Boolean (like true), a String (like "foo"), a Range (like [0: 1: 10]), a Vector (like [1,2,3]), or the Undefined value (undef).

Values can be stored in variables, passed as function arguments, and returned as function results.

[OpenSCAD is a dynamically typed language with a fixed set of data types. There are no type names, and no user defined types. Functions are not values. In fact, variables and functions occupy disjoint namespaces.]

Numbers

Numbers are the most important type of value in OpenSCAD, and they are written in the familiar decimal notation used in other languages. Eg, -1, 42, 0.5, 2.99792458e+8.

[OpenSCAD does not support octal or hexadecimal notation for numbers.]

In addition to decimal numerals, the following names for special numbers are defined:

- **PI**

OpenSCAD has only a single kind of number, which is a 64 bit IEEE floating point number.

[OpenSCAD does not distinguish integers and floating point numbers as two different types, nor does it support complex numbers.]

Because OpenSCAD uses the IEEE floating point standard, there are a few deviations from the behavior of numbers in mathematics:

- We use binary floating point. A fractional number is not represented exactly unless the denominator is a power of 2. For example, 0.2 (2/10) does not have an exact internal representation, but 0.25 (1/4) and 0.125 (1/8) are represented exactly.
- The largest representable number is about 1e308. If a numeric result is too large,

- then the result can be infinity (printed as `inf` by `echo`).
- The smallest representable number is about `-1e308`. If a numeric result is too small, then the result can be `-infinity` (printed as `-inf` by `echo`).
- If a numeric result is invalid, then the result can be Not A Number (printed as **`nan`** by `echo`).
- If a non-zero numeric result is too close to zero to be representable, then the result is `-0` if the result is negative, otherwise it is `0`. Zero (`0`) and negative zero (`-0`) are treated as two distinct numbers by some of the math operations, and are printed differently by `'echo'`, although they compare as equal.

The constants **`'inf'`** and **`'nan'`** are not supported as numeric constants by OpenSCAD, even though you can compute numbers that are printed this way by `'echo'`. You can define variables with these values by using:

```
inf = 1e200 * 1e200;
nan = 0 / 0;
echo(inf, nan);
```

The value `'nan'` is the only OpenSCAD value that is not equal to any other value, including itself. Although you can test if a variable `'x'` has the undefined value using `'x == undef'`, you can't use `'x == 0/0'` to test if `x` is Not A Number. Instead, you must use `'x != x'` to test if `x` is `nan`.

Created with the Personal Edition of HelpNDoc: [Free PDF documentation generator](#)

Boolean Values

Booleans are truth values. There are two Boolean values, namely `true` and `false`. A Boolean is passed as the argument to conditional statement `'if()'`, conditional operator `'?:'`, and logical operators `'!'` (not), `'&&'` (and), and `'||'` (or). In all of these contexts, you can actually pass any quantity. Most values are converted to `'true'` in a Boolean context, the values that count as `'false'` are:

- `false`
- `0` and `-0`
- `""`
- `[]`
- `undef`

Note that `"false"` (the string), `[0]` (a numeric vector), `[[]]` (a vector containing an empty vector), `[false]` (a vector containing the Boolean value `false`) and `0/0` (Not A Number) all count as `true`.

Created with the Personal Edition of HelpNDoc: [Easily create EPub books](#)

Strings

A string is a sequence of zero or more unicode characters. String values are used to specify file names when importing a file, and to display text for debugging purposes when using `echo()`.

Strings can also be used with the [text\(\) primitive](#).

A string literal is written as a sequence of characters enclosed in quotation marks `"`, like this: `""` (an empty string), or `"this is a string"`.

To include a `"` character in a string literal, use `\`. To include a `\` character in a string literal, use `\\`. The following escape sequences beginning with `\` can be used within string literals:

- `\"` → `"`

- o `\\` → `\`
- o `\t` → `tab`
- o `\n` → `newline`
- o `\r` → `carriage return`
- o `\u03a9` → - see `text()` for further information on unicode characters

Created with the Personal Edition of HelpNDoc: [Produce online help for Qt applications](#)

Ranges

Ranges are used by `for()` loops and `children()`. They have 2 varieties:

```
[<start>:<end>]
[<start>:<increment>:<end>]
```

Although enclosed in square brackets `[]`, they are not vectors. They use colons `:` for separators rather than commas.

```
r1 = [0:10];
r2 = [0.5:2.5:20];
echo(r1); // ECHO: [0: 1: 10]
echo(r2); // ECHO: [0.5: 2.5: 20]
```

You should avoid step values that cannot be represented exactly as binary floating point numbers. Integers are okay, as are fractional values whose denominator is a power of two. For example, 0.25 (1/4) and 0.125 (1/8) are safe, but 0.2 (2/10) should be avoided. The problem with these step values is that your range may have too many or too few elements, due to inexact arithmetic.

A missing *<increment>* defaults to 1.

A range in the form `[<start>:<end>]` with *<start>* greater than *<end>* generates a warning and is equivalent to `[<end>: 1: <start>]`.

A range in the form `[<start>:1:<end>]` with *<start>* greater than *<end>* does not generate a warning and is equivalent to `[]`.

The *<increment>* in a range may be negative.

Created with the Personal Edition of HelpNDoc: [Single source CHM, PDF, DOC and HTML Help creation](#)

The Undefined Value

The undefined value is a special value written as **undef**. It's the initial value of a variable that hasn't been assigned a value, and it is often returned as a result by functions or operations that are passed illegal arguments. Finally, **undef can be used as a null value**, equivalent to `null` or `NULL` in other programming languages.

All arithmetic expressions containing `undef` values evaluate as `undef`.

In logical expressions, `undef` is equivalent to `false`.

Relational operator expressions with `undef` evaluate as `false` except for `undef==undef` which is `true`.

Note that numeric operations may also return 'nan' (not-a-number) to indicate an illegal argument. For example, `0/false` is `undef`, but `0/0` is 'nan'. Relational operators like `<` and `>`

return `false` if passed illegal arguments. Although `undef` is a language value, 'nan' is not.

Created with the Personal Edition of HelpNDoc: [Create help files for the Qt Help Framework](#)

Variables

OpenSCAD variables are created by a statement with a name or **identifier**, assignment via an expression and a semicolon. The role of arrays, found in many imperative languages, is handled in OpenSCAD via vectors.

```
var = 25;
xx = 1.25 * cos(50);
y = 2 * xx + var;
logic = true;
MyString = "This is a string";
a_vector = [1,2,3];
rr = a_vector[2]; // member of vector
range1 = [-1.5:0.5:3]; // for() loop range
xx = [0:5]; // alternate for() loop range
```

OpenSCAD is a **Functional** programming language, as such **variables** are bound to expressions and keep a single value during their entire lifetime due to the requirements of **referential transparency**. In **imperative languages**, such as C, the same behavior is seen as constants, which are typically contrasted with normal variables.

In other words OpenSCAD variables are more like constants, but with an important difference. If variables are assigned a value multiple times, only the last assigned value is used in all places in the code.

See further discussion at [Variables are set at compile-time, not run-time](#). This behavior is due to the need to supply variable input on the [command line](#), via the use of `-D variable=value` option. OpenSCAD currently places that assignment at the end of the source code, and thus must allow a variable's value to be changed for this purpose.

Values cannot be modified during run time; all variables are effectively constants that do not change. Each variable retains its last assigned value at compile time, in line with **Functional** programming languages. Unlike **Imperative** languages, such as C, OpenSCAD is not an iterative language, and as such **the concept of $x = x + 1$ is not valid**. Understanding this concept leads to understanding the beauty of OpenSCAD.

Variables can be assigned in any scope. Note that assignments are only valid within the scope in which they are defined - you are still not allowed to leak values to an outer scope. See [Scope of variables](#) for more details.

```
a=0;
if (a==0) {
    a=1; // but the value a=1 is confined to within the braces {}
}
```

Created with the Personal Edition of HelpNDoc: [Easy to use tool to create HTML Help files and Help web sites](#)

Undefined variable

A non assigned variable has the special value **undef**. It could be tested in conditional expression, and returned by a function.

```
Example
echo("Variable a is ", a); // Variable a is undef
```

```

if (a == undef) {
    echo("Variable a is tested undefined"); // Variable a is tested
    undefined
}

```

Created with the Personal Edition of HelpNDoc: [Create cross-platform Qt Help files](#)

Scope of variables

When operators such as `translate()` and `color()` need to encompass more than one action (actions end in `;`), braces `{}` are needed to group the actions, creating a new, inner scope. When there is only one semicolon, braces are usually optional.

Each pair of braces creates a new scope inside the scope where they were used. New variables can be created within this new scope. New values can be given to variables which were created in an outer scope. These variables and their values are also available to further inner scopes created within this scope, but are **not available** to any thing outside this scope. Variables still have only the last value assigned within a scope.

```

                                // scope 1
a = 6;                          // create a
echo(a, b);                     //           6, undef
translate([5, 0, 0]){          // scope 1.1
    a = 10;
    b = 16;                     // create b
    echo(a, b);                 //           100, 16 a=10; was overridden by
later a = 100;
    color("blue") {             // scope 1.1.1
        echo(a, b);             //           100, 20
        cube();
        b=20;
    }                           // back to 1.1
    echo(a, b);                 // 100, 16
    a = 100;                    // override a in 1.1
}                                // back to 1
echo(a, b);                     // 6, undef
color("red"){                   // scope 1.2
    cube();
    echo(a,b);                  // 6, undef
}                                // back to 1
echo(a, b);                     // 6, undef

```

//In this example, scopes 1 and 1.1 are outer scopes to 1.1.1 but 1.2 is not.

Anonymous scopes are not considered scopes:

```
{ angle = 45; } rotate(angle) square(10);
```

For() loops are not an exception to the rule about variables having only one value within a scope. A copy of loop contents is created for each pass. Each pass is given its own scope, allowing any variables to have unique values for that pass. No, you still can't do `a=a+1`;

Created with the Personal Edition of HelpNDoc: [Create help files for the Qt Help Framework](#)

Variables are set at compile-time, not at run-time

Because OpenSCAD calculates its variable values at compile-time, not run-time, **the last variable assignment, within a scope apply everywhere in that scope, or inner scopes thereof**. It may be helpful to think of them as override-able constants rather than as variables.

```
// The value of 'a' reflects only the last set value
a = 0;
echo(a); // 5
a = 3;
echo(a); // 5 a = 5;
```

While this appears to be counter-intuitive, it allows you to do some interesting things: for instance, if you set up your shared library files to have default values defined as variables at their root level, when you include that file in your own code, you can 're-define' or override those constants by simply assigning a new value to them.

Created with the Personal Edition of HelpNDoc: [News and information about help authoring tools and software](#)

Special Variables

Special variables provide an alternate means of passing arguments to modules and functions. **All variables starting with a '\$' are special variables**, similar to special variables in lisp. As such they are more dynamic than regular variables. (for more details see [Other Language Features](#))

Created with the Personal Edition of HelpNDoc: [Benefits of a Help Authoring Tool](#)

Vectors

A vector is a sequence of zero or more OpenSCAD values. Vectors are a collection (or list or table) of numeric or boolean values, variables, vectors, strings or any combination thereof. They can also be expressions which evaluate to one of these.

Vectors handle the role of arrays found in many imperative languages. The information here also applies to lists and tables which use vectors for their data.

A vector has square brackets, [] enclosing zero or more items (elements or members), separated by commas. A vector can contain vectors, which contain vectors, etc.

examples

```
[1, 2, 3]
[a, 5, b]
[]
[5.643]
["a", "b", "string"]
[[1, r], [x, y, z, 4, 5]]
[3, 5, [6,7], [[8, 9], [10, [11, 12], 13], c, "string"]
[4/3, 6*1.5, cos(60)]
```

use in OpenSCAD:

```
cube( [width, depth, height]); // optional spaces shown for clarity
translate( [x, y, z] )
```

```
polygon( [ [x0, y0], [x1, y1], [x2, y2] ] );
```

creation

Vectors are created by writing the list of elements, separated by commas, and enclosed in square brackets. Variables are replaced by their values.

```
cube([10, 15, 20]);
a1 = [1, 2, 3];
a2 = [4, 5];
a3 = [6, 7, 8, 9];
b = [a1, a2, a3]; // [ [1,2,3], [4,5], [6,7,8,9] ] note increased nesting depth
```

elements within vectors

Elements within vectors are numbered from 0 to n-1 where n is the length returned by `len()`. Address elements within vectors with the following notation:

```
e[5]           // element no 5 (sixth) at 1st nesting level
e[5][2]        // element 2 of element 5 2nd nesting level
e[5][2][0]     // element 0 of 2 of 5 3rd nesting level
e[5][2][0][1]  // element 1 of 0 of 2 of 5 4th nesting level
```

example elements with lengths from `len()`

```
e = [ [1], [], [3, 4, 5], "string", "x", [[10, 11], [12, 13, 14], [[15, 16], [17]]] ]; // length 6
```

address	length	element
e[0]	1	[1]
e[1]	0	[]
e[5]	3	[[10,11], [12,13,14], [[15,16],[17]]]
e[5][1]	3	[12, 13, 14]
e[5][2]	2	[[15, 16], [17]]
e[5][2][0]	2	[15, 16]
e[5][2][0][1]	undef	16
e[3]	6	"string"
e[3][2]	1	"r"
s = [2,0,5];	a = 2;	
s[a]	undef	5
e[s[a]]	3	[[10, 11], [12, 13, 14], [[15, 16], [17]]]

alternate dot notation

The first three elements of a vector can be accessed with an alternate dot notation:

```
e.x //equivalent to e[0]
e.y //equivalent to e[1]
e.z //equivalent to e[2]
```

Vector Operators (concat & len)

concat

`concat()` combines the elements of 2 or more vectors into a single vector. No change in nesting level is made.

```
vector1 = [1, 2, 3]; vector2 = [4]; vector3 = [5,6];
new_vector = concat(vector1, vector2, vector3); // [1, 2, 3, 4, 5, 6]

string_vector = concat("abc", "def"); // ["abc", "def"]
one_string = str(string_vector[0], string_vector[1]); // "abcdef"
```

len

`len()` returns the length of vectors or strings. Indices of elements are from [0] to [length-1].

vector

- Returns the number of elements at this level.
- Single values, which are **not** vectors, return **undef**.

string

- Returns the number of characters in string.

```
a = [1, 2, 3]; echo(len(a)); // 3
```

Created with the Personal Edition of HelpNDoc: [Create cross-platform Qt Help files](#)

Matrix

A matrix is a vector of vectors.

```
Example which defines a 2D rotation matrix
mr = [
    [cos(angle), -sin(angle)],
    [sin(angle), cos(angle)]
];
```

Created with the Personal Edition of HelpNDoc: [Free CHM Help documentation generator](#)

Getting input

Now we have variables, it would be nice to be able to get input into them instead of setting the values from code. There are a few functions to read data from DXF files, or you can set a variable with the -D switch on the command line.

Getting a point from a drawing

Getting a point is useful for reading an origin point in a 2D view in a technical drawing. The function `dx_f_cross` reads the intersection of two lines on a layer you specify and returns the intersection point. This means that the point must be given with two lines in the DXF file, and not a point entity.

```
OriginPoint = dx_f_cross(file    = "drawing.dxf",
                        layer    = "SCAD.Origin",
                        origin    = [0, 0], scale = 1);
```

Getting a dimension value

You can read dimensions from a technical drawing. This can be useful to read a rotation angle,

an extrusion height, or spacing between parts. In the drawing, create a dimension that does not show the dimension value, but an identifier. To read the value, you specify this identifier from your program:

```
TotalWidth = dxf_dim(file    = "drawing.dxf",
                      name    = "TotalWidth",
                      layer    = "SCAD.Origin",
                      origin   = [0, 0],
                      scale    = 1);
```

For a nice example of both functions, see Example009 and the image on the [homepage of OpenSCAD](#).

Created with the Personal Edition of HelpNDoc: [Easily create EBooks](#)

3D Objects

Created with the Personal Edition of HelpNDoc: [Generate Kindle eBooks with ease](#)

Primitive Solids

Created with the Personal Edition of HelpNDoc: [Produce Kindle eBooks easily](#)

cube

Creates a cube in the first octant. When center is true, the cube is centered on the origin. Argument names are optional if given in the order shown here.

```
cube(size = [x,y,z], center = true/false); cube(size = x , center
= true/false);
```

parameters:

size

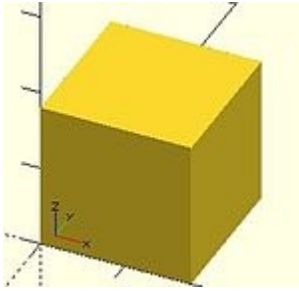
single value, cube with all sides this length
3 value array [x,y,z], cube with dimensions x, y and z.

center

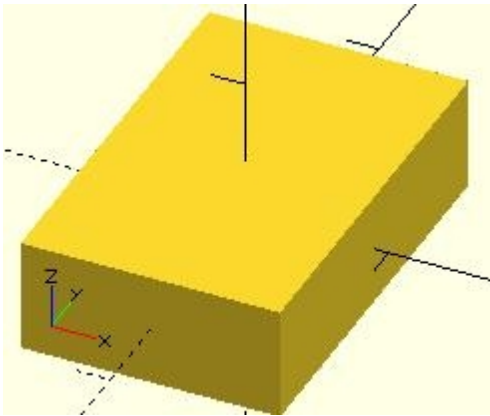
false (default), 1st (positive) octant, one corner at (0,0,0)
true, cube is centered at (0,0,0)

```
default values: cube(); yields: cube(size = [1, 1, 1], center =
false);
```

examples:



```
equivalent scripts for this example cube(size = 18); cube(18);
cube([18,18,18]); . cube(18,false); cube([18,18,18],false);
cube([18,18,18],center=false); cube(size = [18,18,18], center =
false); cube(center = false,size = [18,18,18] );
```



```
equivalent scripts for this example cube([18,28,8],true);
box=[18,28,8];cube(box,true);
```

Created with the Personal Edition of HelpNDoc: [Free Web Help generator](#)

sphere

Creates a sphere at the origin of the coordinate system. The *r* argument name is optional. To use *d* instead of *r*, *d* must be named.

Parameters

r

Radius. This is the radius of the sphere. The resolution of the sphere is based on the size of the sphere and the *\$fa*, *\$fs* and *\$fn* variables. For more information on these special variables look at: [OpenSCAD_User_Manual/Other_Language_Features](#)

d

Diameter. This is the diameter of the sphere.

\$fa

Fragment angle in degrees

\$fs

Fragment size in mm

\$fn

Resolution

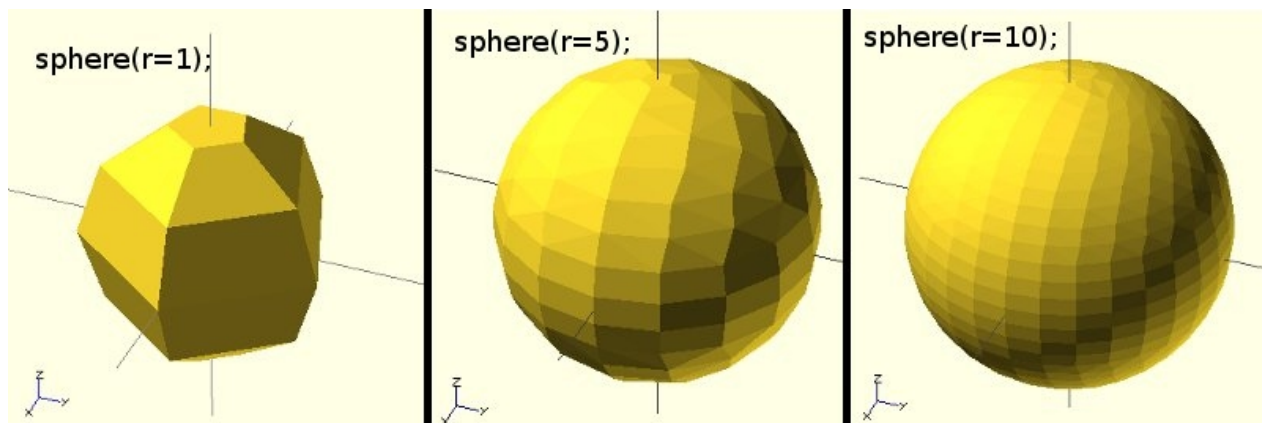
```
default values: sphere(); yields: sphere($fn = 0, $fa = 12, $fs = 2, r = 1);
```

Usage Examples

```
sphere(r = 1); sphere(r = 5); sphere(r = 10); sphere(d = 2);  
sphere(d = 10); sphere(d = 20);
```

```
// this creates a high resolution sphere with a 2mm radius  
sphere(2, $fn=100);
```

```
// also creates a 2mm high resolution sphere but this one // does  
not have as many small triangles on the poles of the sphere  
sphere(2, $fa=5, $fs=0.1);
```



Created with the Personal Edition of HelpNDoc: [Easily create EPub books](#)

cylinder

Creates a cylinder or cone centered about the z axis. When center is true, it is also centered vertically along the z axis.

Parameter names are optional if given in the order shown here. If a parameter is named, all following parameters must also be named.

NOTE: If r, d, d1 or d2 are used they must be named.

```
cylinder(h = height, r1 = BottomRadius, r2 = TopRadius, center =  
true/false);
```

Parameters

h : height of the cylinder or cone

r : radius of cylinder. $r1 = r2 = r$.

r1 : radius, bottom of cone.

r2 : radius, top of cone.

d : diameter of cylinder. $r1 = r2 = d / 2$. **[Note: Requires version 2014.03]**

d1 : diameter, bottom of cone. $r1 = d1 / 2$. **[Note: Requires version 2014.03]**

d2 : diameter, top of cone. $r2 = d2 / 2$. **[Note: Requires version 2014.03]**

center

false (default), z ranges from 0 to h

true, z ranges from -h/2 to +h/2

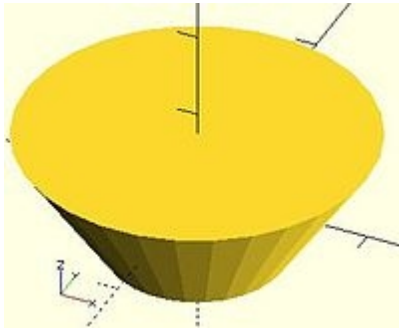
\$fa : minimum angle (in degrees) of each fragment.

\$fs : minimum circumferential length of each fragment.

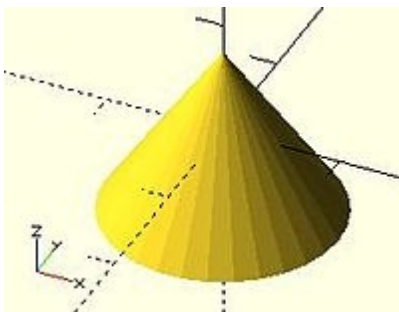
\$fn : **fixed** number of fragments in 360 degrees. Values of 3 or more override \$fa and \$fs

\$fa, \$fs and \$fn must be named. [click here for more details](#),.

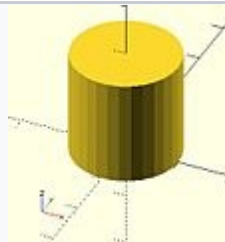
```
defaults: cylinder(); yields: cylinder($fn = 0, $fa = 12, $fs = 2,
h = 1, r1 = 1, r2 = 1, center = false);
```



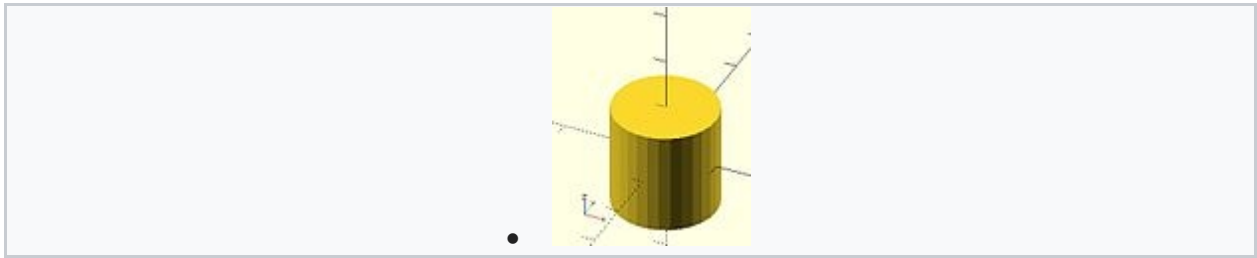
```
equivalent scripts cylinder(h=15, r1=9.5, r2=19.5, center=false);
cylinder( 15, 9.5, 19.5, false); cylinder( 15, 9.5, 19.5);
cylinder( 15, 9.5, d2=39 ); cylinder( 15, d1=19, d2=39 );
cylinder( 15, d1=19, r2=19.5);
```



```
equivalent scripts cylinder(h=15, r1=10, r2=0, center=true);
cylinder( 15, 10, 0, true); cylinder(h=15, d1=20, d2=0,
center=true);
```



center = false



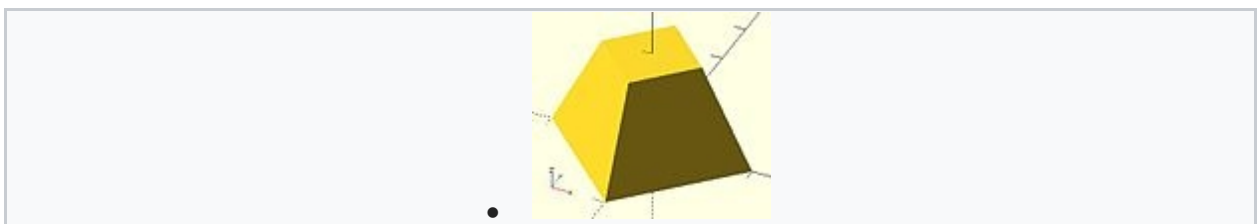
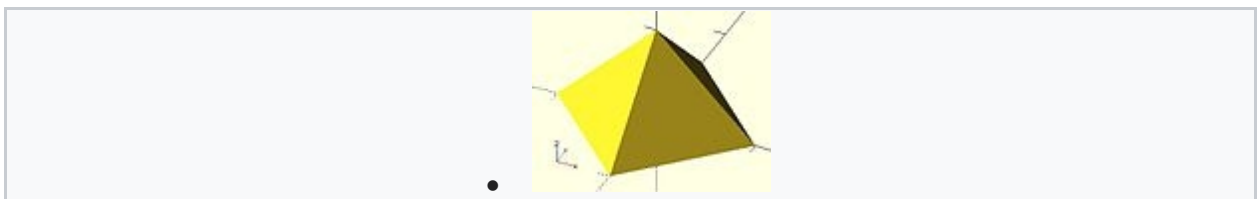
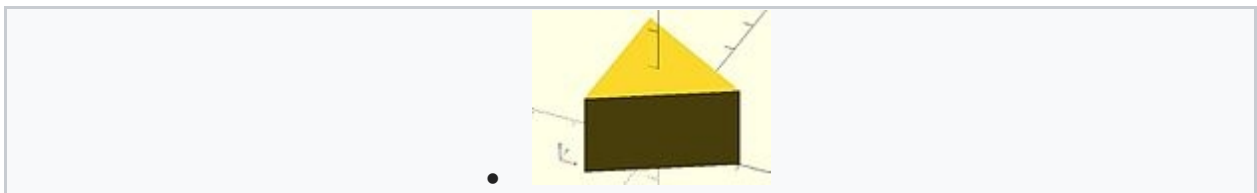
center = true

```
equivalent scripts cylinder(h=20, r=10, center=true);
cylinder( 20, 10, 10,true); cylinder( 20, d=20, center=true);
cylinder( 20,r1=10, d2=20, center=true); cylinder( 20,r1=10,
d2=2*10, center=true);
```

use of \$fn

Larger values of \$fn create smoother, more circular, surfaces at the cost of longer rendering time. Some use medium values during development for the faster rendering, then change to a larger value for the final F6 rendering.

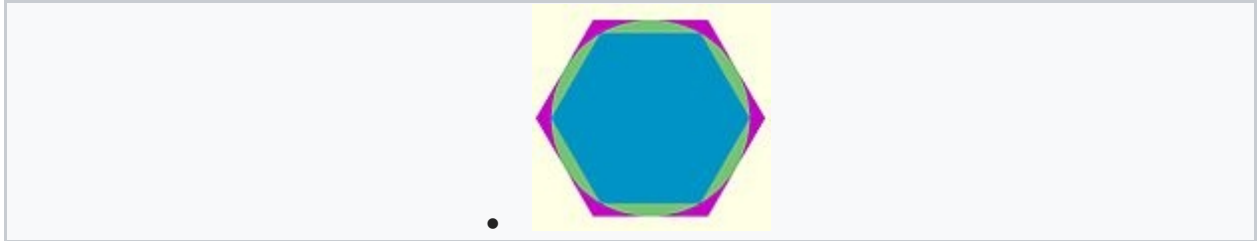
However, use of small values can produce some interesting non circular objects. A few examples are show here:



```
scripts for these examples cylinder(20,20,20,$fn=3);
cylinder(20,20,00,$fn=4); cylinder(20,20,10,$fn=4);
```

undersized holes

Using `cylinder()` with `difference()` to place holes in objects creates undersized holes. This is because circular paths are approximated with polygons inscribed within in a circle. The points of the polygon are on the circle, but straight lines between are inside. To have all of the hole larger than the true circle, the polygon must lie wholly outside of the circle (circumscribed). [Modules for circumscribed holes](#)



Notes on accuracy Circle objects are approximated. The algorithm for doing this matters when you want 3d printed holes to be the right size. Current behavior is [illustrated in a diagram](#) . Discussion regarding optionally changing this behavior happening in a [Pull Request](#)

Created with the Personal Edition of HelpNDoc: [Full-featured Documentation generator](#)

polyhedron

A polyhedron is the most general 3D primitive solid. It can be used to create any regular or irregular shape including those with concave as well as convex features. Curved surfaces are approximated by a series of flat surfaces.

```
polyhedron( points = [ [X0, Y0, Z0], [X1, Y1, Z1], ... ], triangles
= [ [P0, P1, P2], ... ], convexity = N); // before 2014.03
polyhedron( points = [ [X0, Y0, Z0], [X1, Y1, Z1], ... ], faces =
[ [P0, P1, P2, P3, ...], ... ], convexity = N); // 2014.03 & later
```

Parameters

points

Vector of 3d points or vertices. Each point is in turn a vector, `[x,y,z]`, of its coordinates. Points may be defined in any order. N points are referenced, in the order defined, as 0 to N-1.

triangles [**Deprecated:** *triangles will be removed in future releases. Use **faces** parameter instead*]

Vector of faces that collectively enclose the solid. Each face is a vector containing the indices (0 based) of 3 points from the points vector.

faces [**Note:** *Requires version **2014.03***]

Vector of faces that collectively enclose the solid. Each face is a vector containing the indices (0 based) of 3 or more points from the points vector.

Faces may be defined in any order. Define enough faces to fully enclose the solid, with no overlap.

If points that describe a single face are not on the same plane, the face is automatically split into triangles as needed.

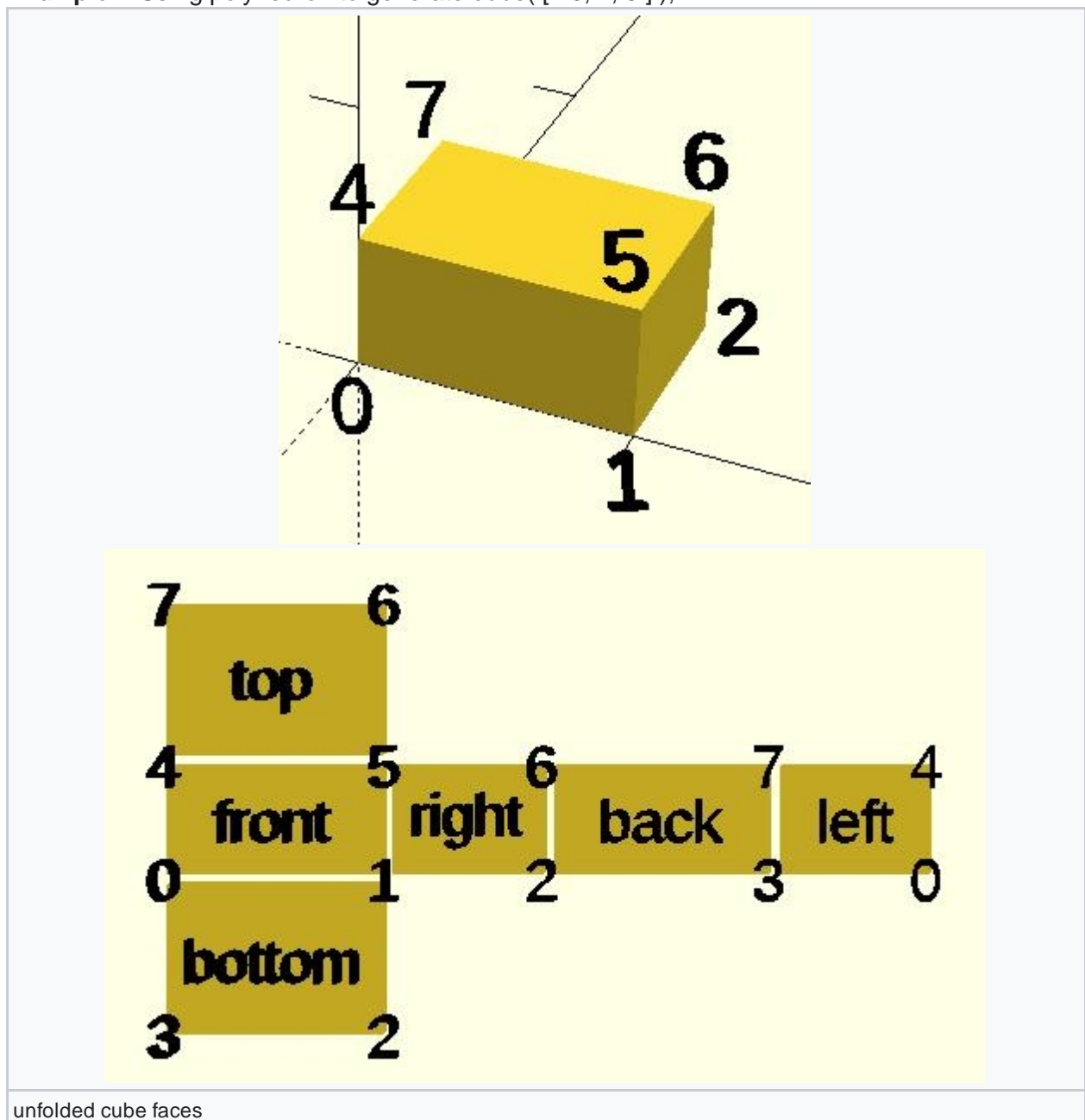
convexity

Integer. The convexity parameter specifies the maximum number of faces a ray intersecting the object might penetrate. This parameter is needed only for correct display of the object in OpenCSG preview mode. It has no effect on the polyhedron rendering. For display problems, setting it to 10 should work fine for most cases.

```
default values: polyhedron(); yields: polyhedron(points = undef,
faces = undef, convexity = 1);
```

All faces must have points ordered in the same direction . OpenSCAD prefers **clockwise** when looking at each face from outside **inward**. The back is viewed from the back, the bottom from the bottom, etc.

Example 1 Using polyhedron to generate cube([10, 7, 5]);



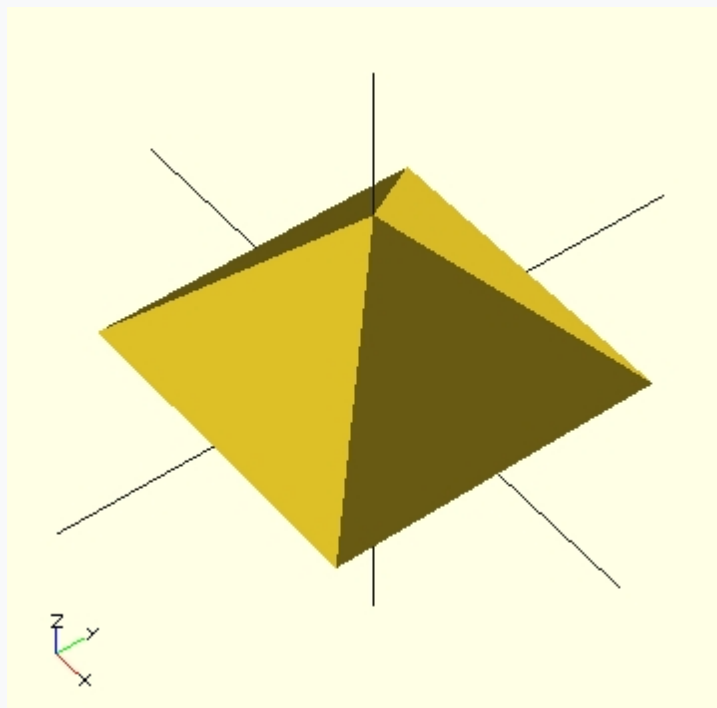
```

CubePoints = [ [ 0, 0, 0 ], //0 [ 10, 0, 0 ], //1 [ 10, 7,
0 ], //2 [ 0, 7, 0 ], //3 [ 0, 0, 5 ], //4 [ 10, 0, 5 ], //5 [ 10,
7, 5 ], //6 [ 0, 7, 5 ]]; //7 CubeFaces = [ [0,1,2,3], // bottom
[4,5,1,0], // front [7,6,5,4], // top [5,6,2,1], // right
[6,7,3,2], // back [7,4,0,3]]; // left polyhedron( CubePoints,
CubeFaces );

```

equivalent descriptions of the bottom face [0,1,2,3], [0,1,2,3,0], [1,2,3,0], [2,3,0,1], [3,0,1,2], [0,1,2],[2,3,0], // 2 triangles with no overlap [1,2,3],[3,0,1], [1,2,3],[0,1,3],

Example 2 A square base pyramid:



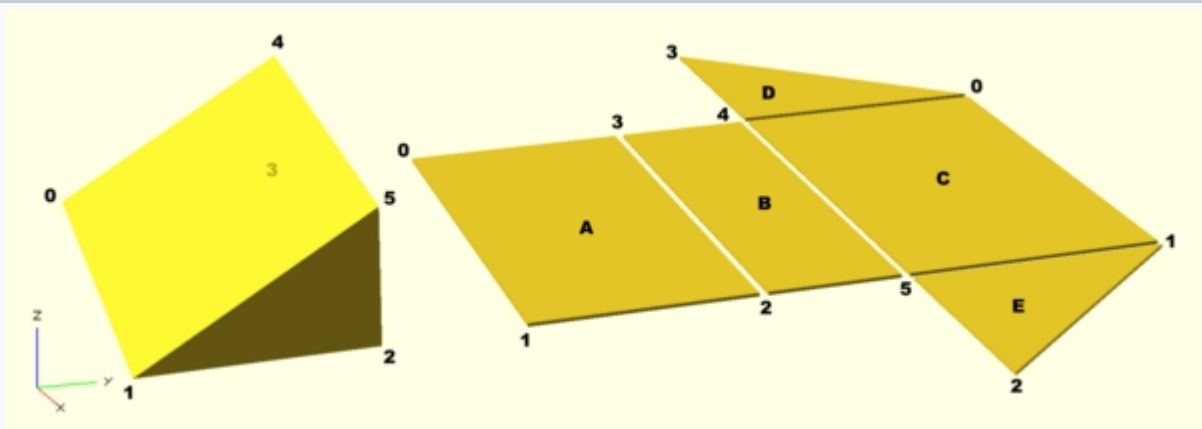
A simple polyhedron, square base pyramid

```

polyhedron( points=[ [10,10,0],[10,-10,0],[-10,-10,0],[-
10,10,0], // the four points at base [0,0,10] ], // the apex point
faces=[ [0,1,4],[1,2,4],[2,3,4],[3,0,4], // each triangle side
[1,0,3],[2,1,3] ] // two triangles for square base );

```

Example 3 A triangular prism:



A polyhedron triangular prism

```

module prism(l, w, h){ polyhedron( points=[[0,0,0], [1,0,0],
[1,w,0], [0,w,0], [0,w,h], [1,w,h]], faces=[[0,1,2,3],[5,4,3,2],
[0,4,5,1],[0,3,4],[5,2,1]] ); // preview unfolded (do not include
in your function z = 0.08; separation = 2; border = .2;
translate([0,w+separation,0]) cube([1,w,z]);
translate([0,w+separation+w+border,0]) cube([1,h,z]);
translate([0,w+separation+w+border+h+border,0])
cube([1,sqrt(w*w+h*h),z]);
translate([1+border,w+separation+w+border+h+border,0])
polyhedron( points=[[0,0,0],[h,0,0],[0,sqrt(w*w+h*h),0], [0,0,z],
[h,0,z],[0,sqrt(w*w+h*h),z]], faces=[[0,1,2], [3,5,4], [0,3,4,1],
[1,4,5,2], [2,5,3,0]] ); translate([0-
border,w+separation+w+border+h+border,0])
polyhedron( points=[[0,0,0],[0-h,0,0],[0,sqrt(w*w+h*h),0],
[0,0,z],[0-h,0,z],[0,sqrt(w*w+h*h),z]], faces=[[1,0,2],[5,3,4],
[0,1,4,3],[1,2,5,4],[2,0,3,5]] ); } prism(10, 5, 3);

```

point numbers for cube

Created with the Personal Edition of HelpNDoc: [Easy EBook and documentation generator](#)

Debugging polyhedra

Mistakes in defining polyhedra include not having all faces with the same order, overlap of faces and missing faces or portions of faces. As a general rule, the polyhedron faces should also satisfy manifold conditions:

- exactly two faces should meet at any polyhedron edge.
- if two faces have a vertex in common, they should be in the same cycle face-edge around the vertex.

The first rule eliminates polyhedra like two cubes with a common edge and not watertight models; the second excludes polyhedra like two cubes with a common vertex.

When viewed from the outside, the points describing each face must be in the same order . OpenSCAD prefers CW, and provides a mechanism for detecting CCW. When the thrown together view (F12) is used with F5, CCW faces are shown in pink. Reorder the points for incorrect faces. Rotate the object to view all faces. The pink view can be turned off with F10.

OpenSCAD allows, temporarily, commenting out part of the face descriptions so that only the remaining faces are displayed. Use // to comment out the rest of the line. Use /* and */ to start and end a comment block. This can be part of a line or extend over several lines. Viewing only part of the faces can be helpful in determining the right points for an individual face. Note that a solid is not shown, only the faces. If using F12, all faces have one pink side. Commenting some faces helps also to show any internal face.

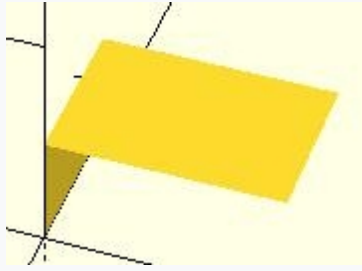
```

CubeFaces = [ /* [0,1,2,3], // bottom [4,5,1,0], // front */
[7,6,5,4], // top /* [5,6,2,1], // right [6,7,3,2], // back */
[7,4,0,3]]; // left

```

After defining a polyhedron, its preview may seem correct. The polyhedron alone may even render fine. However, to be sure it is a valid manifold and that it can generate a valid STL file, union it with any cube and render it (F6). If the polyhedron disappears, it means that it is not

correct. Revise the winding order of all faces and the two rules stated above.



example 1 showing only 2 faces

Created with the Personal Edition of HelpNDoc: [Easily create Web Help sites](#)

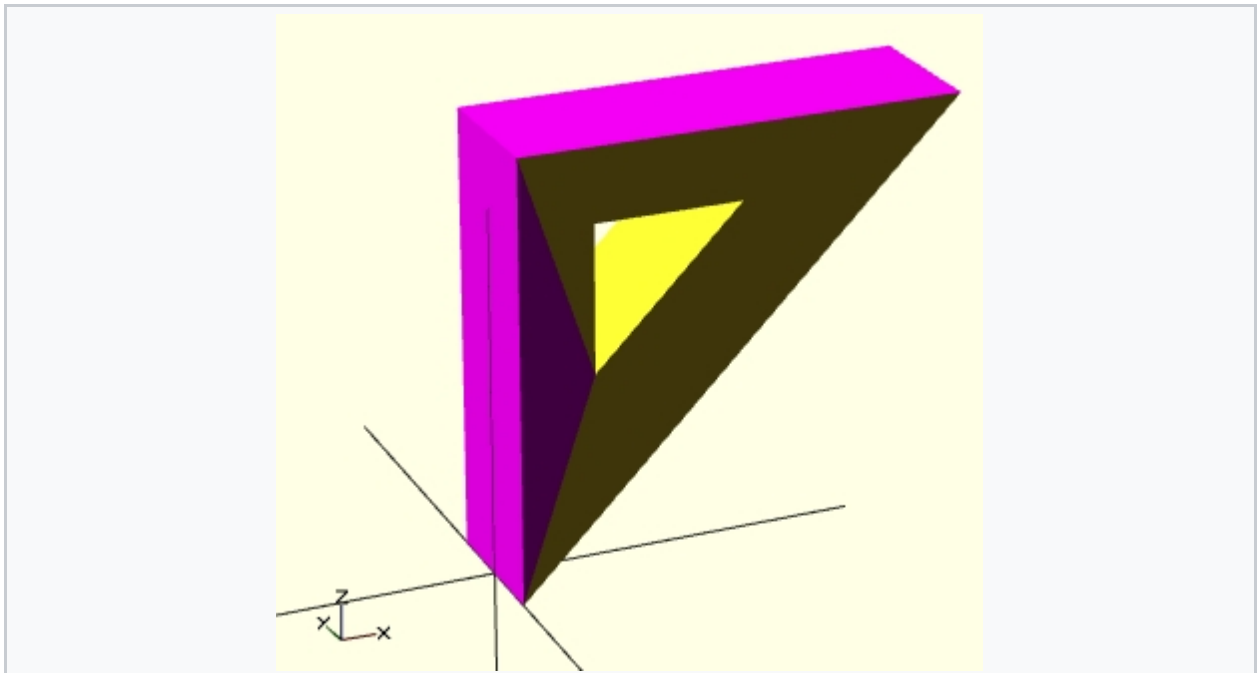
Mis-ordered faces

Example 4 a more complex polyhedron with mis-ordered faces

When you select 'Thrown together' from the view menu and **compile** the design (**not** compile and render!) the preview shows the mis-oriented polygons highlighted. Unfortunately this highlighting is not possible in the OpenCSG preview mode because it would interfere with the way the OpenCSG preview mode is implemented.)

Below you can see the code and the picture of such a problematic polyhedron, the bad polygons (faces or compositions of faces) are in pink.

```
// Bad polyhedron
polyhedron
(points = [
[0, -10, 60], [0, 10, 60], [0, 10, 0], [0, -10, 0], [60, -10, 60], [60,
10, 60], [10, -10, 50], [10, 10, 50], [10, 10, 30], [10, -10, 30], [30, -
10, 50], [30, 10, 50]
], faces = [
[0,2,3], [0,1,2], [0,4,5], [0,5,1], [5,4,2], [2,4,3],
[6,8,9], [6,7,8], [6,10,11], [6,11,7], [10,8,11],
[10,9,8], [0,3,9], [9,0,6], [10,6, 0], [0,4,10],
[3,9,10], [3,10,4], [1,7,11], [1,11,5], [1,7,8], [1,8,2], [2,8,11],
[2,11,5]
]
);
```

Polyhedron with badly oriented polygons

A correct polyhedron would be the following:

```
polyhedron
(points = [
[0, -10, 60], [0, 10, 60], [0, 10, 0], [0, -10, 0], [60, -10, 60], [60,
10, 60], [10, -10, 50], [10, 10, 50], [10, 10, 30], [10, -10, 30], [30, -
10, 50], [30, 10, 50]
], faces = [
[0,3,2], [0,2,1], [4,0,5], [5,0,1], [5,2,4], [4,2,3],
[6,8,9], [6,7,8], [6,10,11],[6,11,7], [10,8,11],
[10,9,8], [3,0,9], [9,0,6], [10,6, 0],[0,4,10],
[3,9,10], [3,10,4], [1,7,11], [1,11,5], [1,8,7], [2,8,1], [8,2,11],
[5,11,2]
]
);
```

Created with the Personal Edition of HelpNDoc: [Easily create Help documents](#)

Point repetitions in a polyhedron point list

Beginner's tip

If you don't really understand "orientation", try to identify the mis-oriented pink faces and then invert the sequence of the references to the points vectors until you get it right. E.g. in the above example, the third triangle ($[0,4,5]$) was wrong and we fixed it as $[4,0,5]$. Remember that a face list is a circular list. In addition, you may select "Show Edges" from the "View Menu", print a screen capture and number both the points and the faces. In our example, the points are annotated in black and the faces in blue. Turn the object around and make a second copy from the back if needed. This way you can keep track.

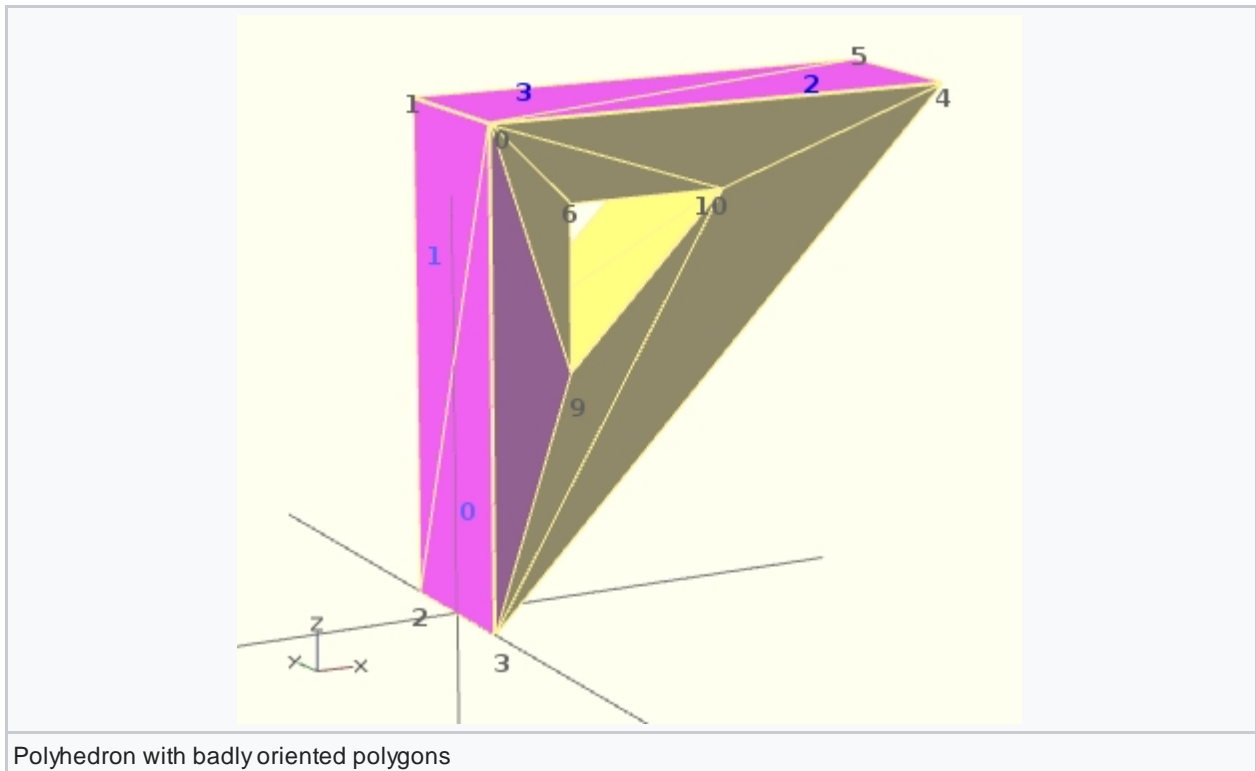
Clockwise Technique

Orientation is determined by clockwise circular indexing. This means that if you're looking at the triangle (in this case $[4,0,5]$) from the outside you'll see that the path is clockwise around the

center of the face. The winding order [4,0,5] is clockwise and therefore good. The winding order [0,4,5] is counter-clockwise and therefore bad. Likewise, any other clockwise order of [4,0,5] works: [5,4,0] & [0,5,4] are good too. If you use the clockwise technique, you'll always have your faces outside (outside of OpenSCAD, other programs do use counter-clockwise as the outside though).

Think of it as a Left Hand Rule:

If you place your left hand on the face with your fingers curled in the direction of the order of the points, your thumb should point outward. If your thumb points inward, you need to reverse the winding order.



Succinct description of a 'Polyhedron'

```
* Points define all of the points/vertices in the shape. * Faces
is a list of flat polygons that connect up the points/vertices.
```

Each point, in the point list, is defined with a 3-tuple x,y,z position specification. Points in the point list are automatically enumerated starting from zero for use in the faces list (0,1,2,3,... etc).

Each face, in the faces list, is defined by selecting 3 or more of the points (using the point order number) out of the point list.

e.g. faces=[[0,1,2]] defines a triangle from the first point (points are zero referenced) to the second point and then to the third point.

When looking at any face from the outside, the face must list all points in a clockwise order.

Point repetitions in a polyhedron point list[\[edit\]](#)

The point list of the polyhedron definition may have repetitions. When two or more points have the same coordinates they are considered the same polyhedron vertex. So, the following polyhedron:

```
points = [[ 0, 0, 0], [10, 0, 0], [ 0,10, 0],
[ 0, 0, 0], [10, 0, 0], [ 0,10, 0],
[ 0,10, 0], [10, 0, 0], [ 0, 0,10],
[ 0, 0, 0], [ 0, 0,10], [10, 0, 0],
[ 0, 0, 0], [ 0,10, 0], [ 0, 0,10]];
polyhedron(points, [[0,1,2], [3,4,5], [6,7,8], [9,10,11], [12,13,14]]);
```

define the same tetrahedron as:

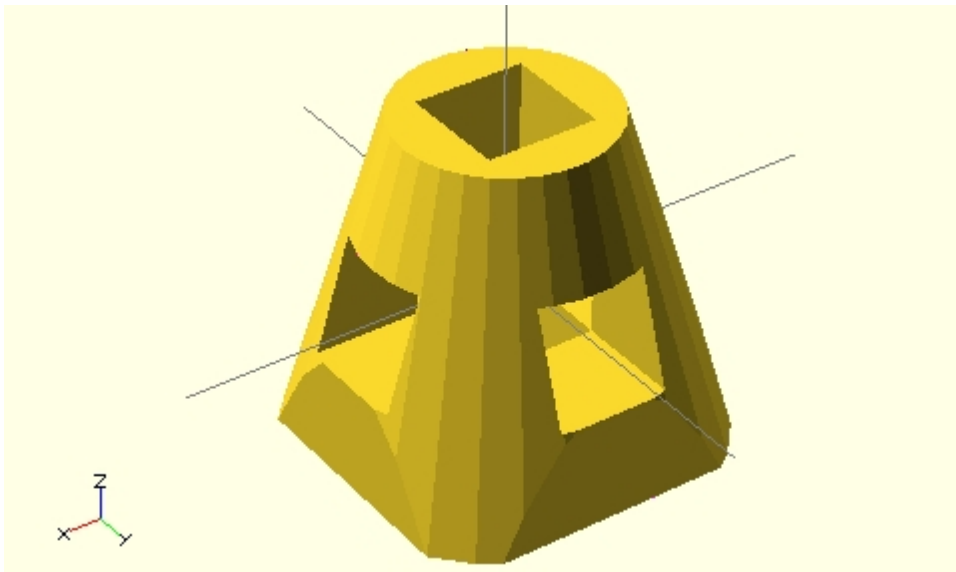
```
points = [[0,0,0], [0,10,0], [10,0,0], [0,0,10]];
polyhedron(points, [[0,2,1], [0,1,3], [1,2,3], [0,3,2]]);
```

Created with the Personal Edition of HelpNDoc: [News and information about help authoring tools and software](#)

3D to 2D Projection

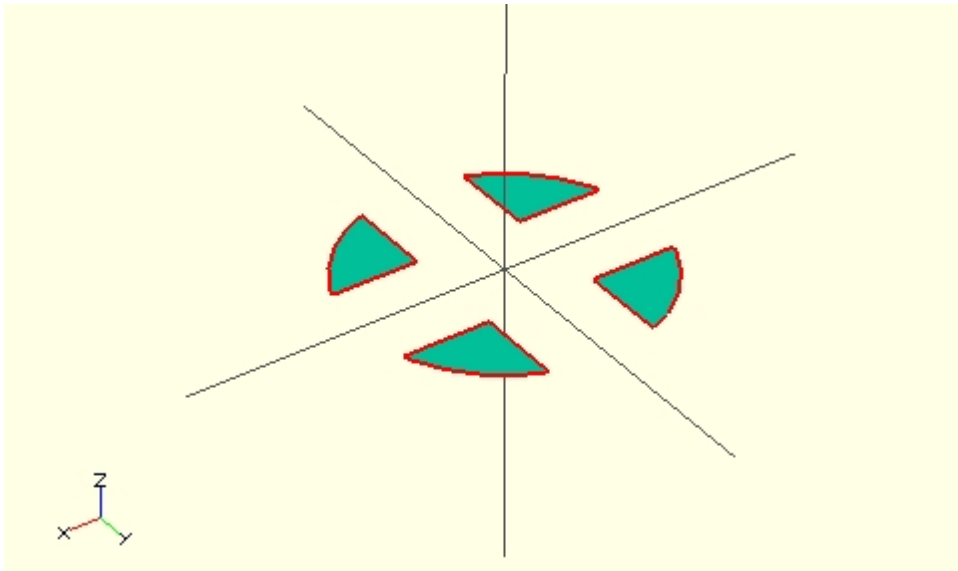
Using the `projection()` function, you can create 2d drawings from 3d models, and export them to the dxf format. It works by projecting a 3D model to the (x,y) plane, with z at 0. If `cut=true`, only points with `z=0` are considered (effectively cutting the object), with `cut=false` (*the default*), points above and below the plane are considered as well (creating a proper projection).

Example: Consider `example002.scad`, that comes with OpenSCAD.



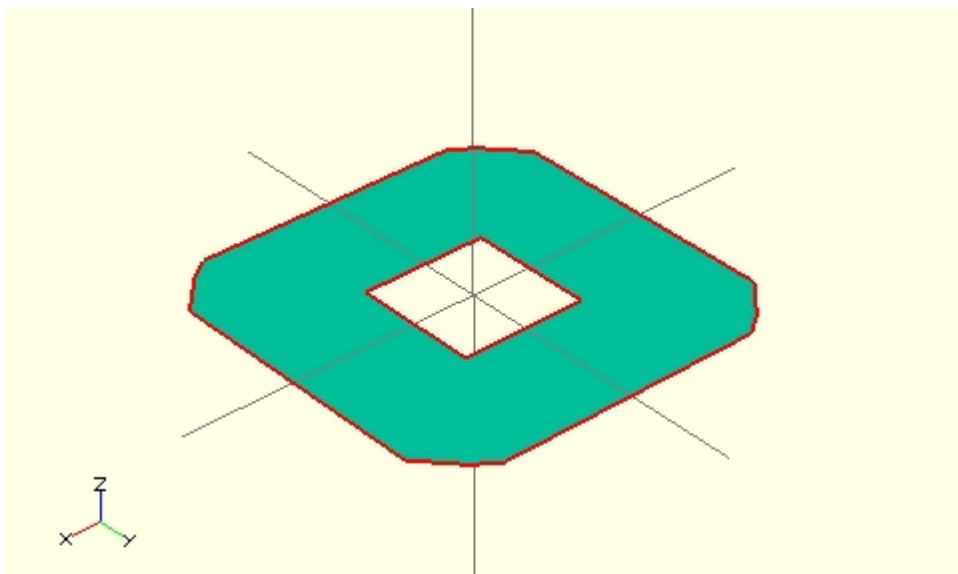
Then you can do a 'cut' projection, which gives you the 'slice' of the x-y plane with `z=0`.

```
projection(cut = true) example002();
```



You can also do an 'ordinary' projection, which gives a sort of 'shadow' of the object onto the xy plane.

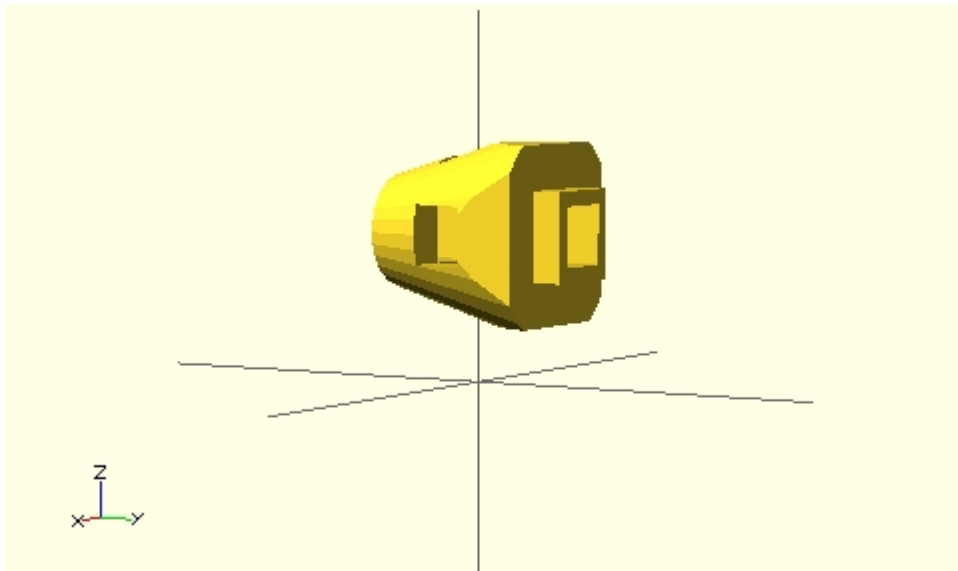
```
projection(cut = false) example002();
```



Another Example

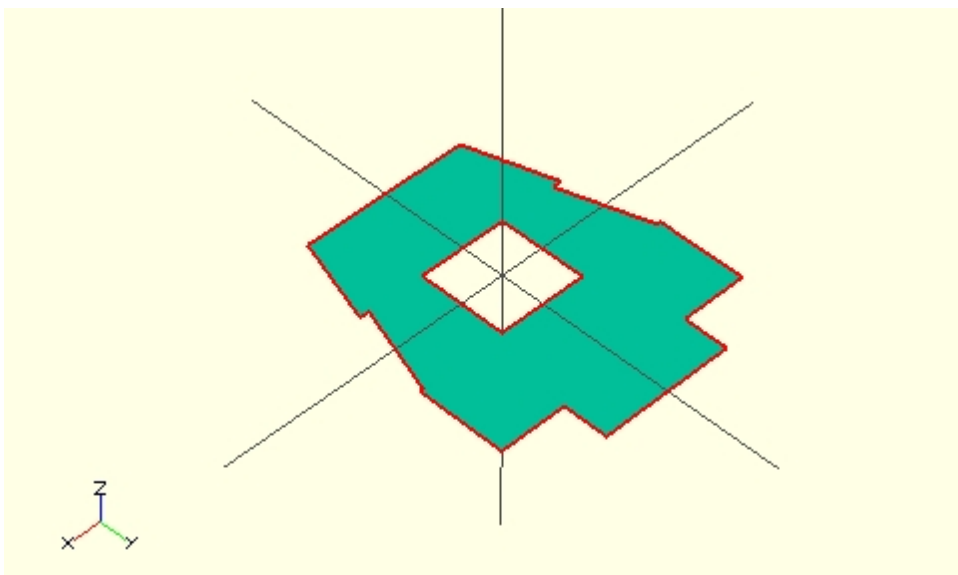
You can also use projection to get a 'side view' of an object. Let's take example002, and move it up, out of the X-Y plane, and rotate it:

```
translate([0,0,25]) rotate([90,0,0]) example002();
```



Now we can get a side view with `projection()`

```
projection() translate([0,0,25]) rotate([90,0,0]) example002();
```



Links:

- [example021.scad](#) from Clifford Wolf's site.
- [More complicated example](#) from Giles Bathgate's blog

Created with the Personal Edition of HelpNDoc: [Full-featured Help generator](#)

2D Objects

All 2D primitives can be transformed with 3D transformations. Usually used as part of a 3D extrusion. Although infinitely thin, they are rendered with a 1 thickness.

Created with the Personal Edition of HelpNDoc: [Single source CHM, PDF, DOC and HTML Help creation](#)

square

Creates a square or rectangle in the first quadrant. When center is true the square is centered on the origin. Argument names are optional if given in the order shown here.

```
square(size = [x, y], center = true/false); square(size = x ,
center = true/false);
```

parameters:

size

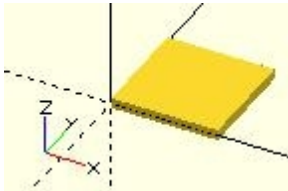
single value, square with both sides this length
2 value array [x,y], rectangle with dimensions x and y

center

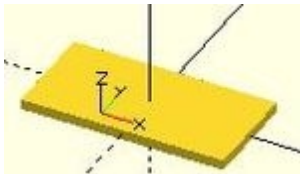
false (default), 1st (positive) quadrant, one corner at (0,0)
true, square is centered at (0,0)

```
default values: square(); yields: square(size = [1, 1], center =
false);
```

examples:



```
equivalent scripts for this example square(size = 10); square(10);
square([10,10]); . square(10,false); square([10,10],false);
square([10,10],center=false); square(size = [10, 10], center =
false); square(center = false,size = [10, 10] );
```



```
equivalent scripts for this example square([20,10],true);
a=[20,10];square(a,true);
```

Created with the Personal Edition of HelpNDoc: [Produce online help for Qt applications](#)

circle

Creates a circle at the origin. All parameters, except r, **must** be named.

```
circle(r=radius | d=diameter);
```

Parameters

r : circle radius. r name is the only one optional with circle.

circle resolution is based on size, using \$fa or \$fs.

For a small, high resolution circle you can make a large circle, then scale it down, or you could set \$fn or other special variables. Note: These examples exceed the resolution of a 3d printer as well as of the display screen.

```
scale([1/100, 1/100, 1/100]) circle(200); // create a high
resolution circle with a radius of 2. circle(2, $fn=50); //
Another way.
```

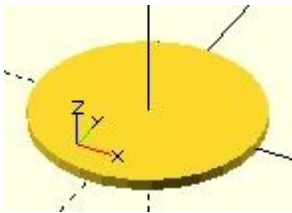
d : circle diameter (only available in versions later than 2014.03).

\$fa : minimum angle (in degrees) of each fragment.

\$fs : minimum circumferential length of each fragment.

\$fn : **fixed** number of fragments in 360 degrees. Values of 3 or more override \$fa and \$fs
\$fa, \$fs and \$fn must be named. [click here for more details](#)..

```
defaults: circle(); yields: circle($fn = 0, $fa = 12, $fs = 2, r =
1);
```

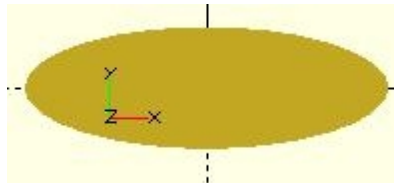
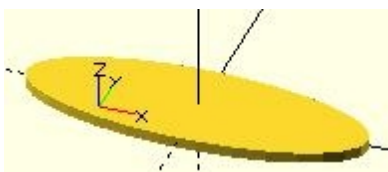


```
equivalent scripts for this example circle(10); circle(r=10);
circle(d=20); circle(d=2+9*2);
```

Created with the Personal Edition of HelpNDoc: [Easily create Web Help sites](#)

ellipse

An ellipse can be created from a circle by using either `scale()` or `resize()` to make the x and y dimensions unequal. See [OpenSCAD User Manual/Transformations](#)



```
equivalent scripts for this example resize([30,10])circle(d=20);
scale([1.5,.5])circle(d=20);
```

Created with the Personal Edition of HelpNDoc: [Write EPub books for the iPad](#)

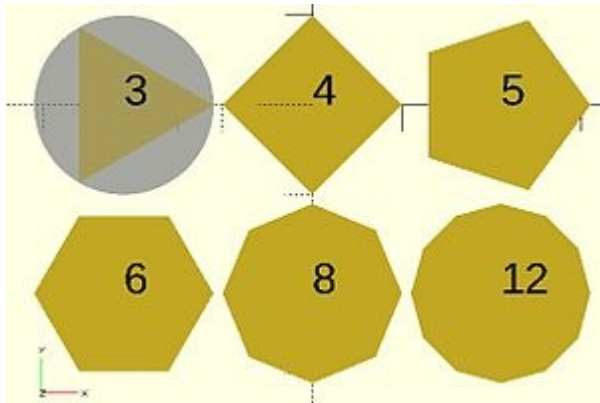
regular polygon

A regular polygon of 3 or more sides can be created by using `circle()` with `$fn` set to the number of sides. The following two pieces of code are equivalent.

```
circle(r=1, $fn=4);
```

```
module regular_polygon(order = 4, r=1){ angles=[ for (i =
[0:order-1]) i*(360/order) ]; coords=[ for (th=angles) [r*cos(th),
r*sin(th)] ]; polygon(coords); } regular_polygon();
```

These result in the following shapes, where the polygon is inscribed within the circle with all sides (and angles) equal. One corner points to the positive x direction. For irregular shapes see the polygon primitive below.



```
script for these examples translate([-42, 0]){circle(20,$fn=3);%
circle(20,$fn=90);} translate([ 0, 0]) circle(20,$fn=4);
translate([ 42, 0]) circle(20,$fn=5); translate([-42,-42])
circle(20,$fn=6); translate([ 0,-42]) circle(20,$fn=8);
translate([ 42,-42]) circle(20,$fn=12);
```

```
color("black"){ translate([-42, 0,1])text("3",7,,center);
translate([ 0, 0,1])text("4",7,,center); translate([ 42, 0,1])
text("5",7,,center); translate([-42,-42,1])text("6",7,,center);
translate([ 0,-42,1])text("8",7,,center); translate([ 42,-42,1])
text("12",7,,center); }
```

Created with the Personal Edition of HelpNDoc: [Full-featured Help generator](#)

polygon

Creates a multiple sided shape from a list of x,y coordinates. A polygon is the most powerful 2D object. It can create anything that circle and squares can, as well as much more. This includes irregular shapes with both concave and convex edges. In addition it can place holes within that shape.

```
polygon(points = [ [x, y], ... ], paths = [ [p1, p2, p3...], ...],
convexity = N);
```

Parameters

points

The list of x,y points of the polygon. : A vector of 2 element vectors.

Note: points are indexed from 0 to n-1.

paths

default

If no path is specified, all points are used in the order listed.

single vector

The order to traverse the points. Uses indices from 0 to n-1. May be in a different order and use all or part, of the points listed.

multiple vectors

Creates primary and secondary shapes. Secondary shapes are subtracted from the

primary shape (like difference). Secondary shapes may be wholly or partially within the primary shape.

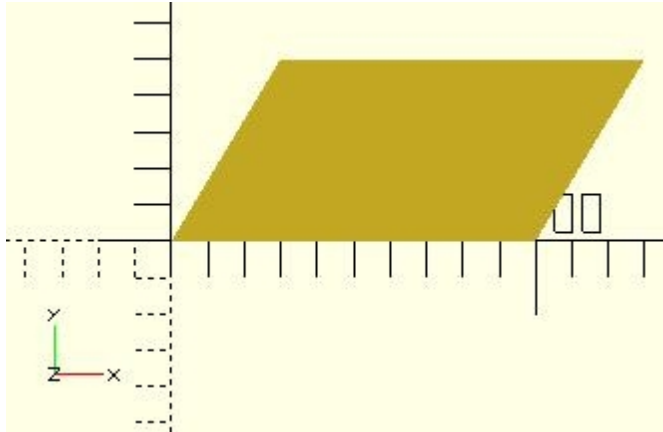
A closed shape is created by returning from the last point specified to the first.

convexity

Integer number of "inward" curves, ie. expected path crossings of an arbitrary line through the polygon. See below.

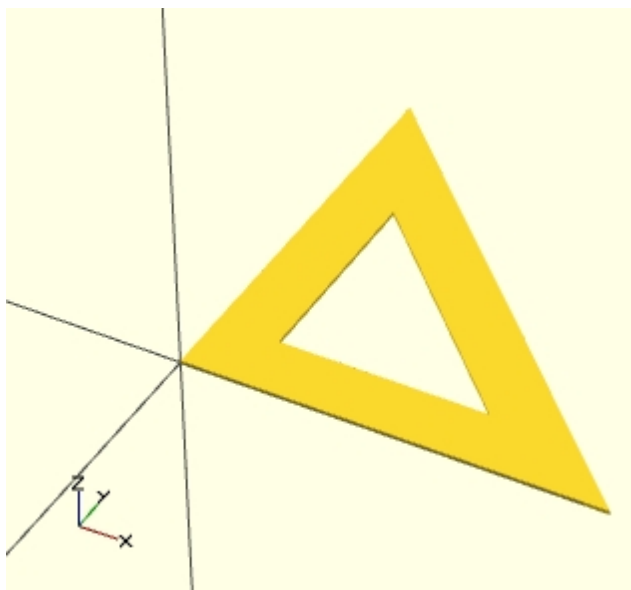
```
defaults: polygon(); yields: polygon(points = undef, paths =
undef, convexity = 1);
```

Example no holes



```
equivalent scripts for this example polygon(points=[[0,0],[100,0],
[130,50],[30,50]]); polygon([[0,0],[100,0],[130,50],[30,50]],
paths=[[0,1,2,3]]); polygon([[0,0],[100,0],[130,50],[30,50]],
[[3,2,1,0]]); polygon([[0,0],[100,0],[130,50],[30,50]],
[[1,0,3,2]]); a=[[0,0],[100,0],[130,50],[30,50]]; b=[[3,0,1,2]];
polygon(a); polygon(a,b); polygon(a,[[2,3,0,1,2]]);
```

Example one hole



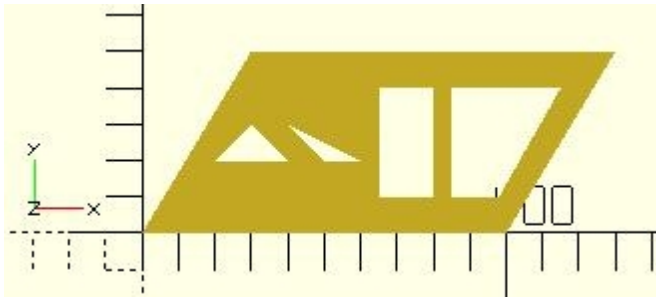
```
equivalent scripts for this example polygon(points=[[0,0],[100,0],
```

```
[0,100],[10,10],[80,10],[10,80]], paths=[[0,1,2],
[3,4,5]],convexity=10); triangle_points =[[0,0],[100,0],[0,100],
[10,10],[80,10],[10,80]]; triangle_paths =[[0,1,2],[3,4,5]];
polygon(triangle_points,triangle_paths,10);
```

The 1st path vector, [0,1,2], selects the points, [0,0],[100,0],[0,100], for the primary shape. The 2nd path vector, [3,4,5], selects the points, [10,10],[80,10],[10,80], for the secondary shape. The secondary shape is subtracted from the primary (think `difference()`). Since the secondary is wholly within the primary, it leaves a shape with a hole.

Example multi hole

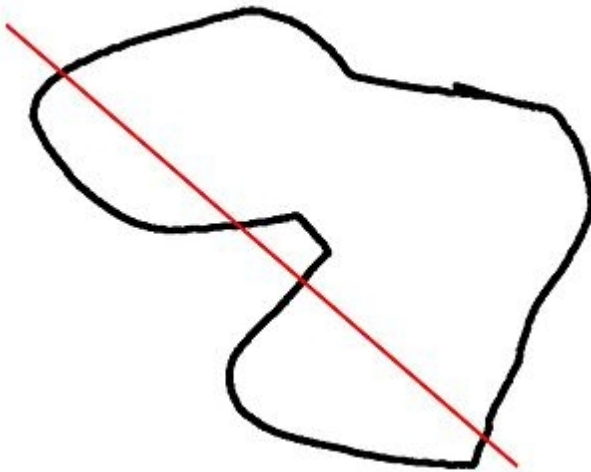
[Note: Requires version 2015.03] (for use of `concat()`)



```
//example polygon with multiple holes a0 = [[0,0],[100,0],
[130,50],[30,50]]; // main b0 = [1,0,3,2]; a1 = [[20,20],[40,20],
[30,30]]; // hole 1 b1 = [4,5,6]; a2 = [[50,20],[60,20],
[40,30]]; // hole 2 b2 = [7,8,9]; a3 = [[65,10],[80,10],[80,40],
[65,40]]; // hole 3 b3 = [10,11,12,13]; a4 = [[98,10],[115,40],
[85,40],[85,10]]; // hole 4 b4 = [14,15,16,17]; a = concat
(a0,a1,a2,a3,a4); b = [b0,b1,b2,b3,b4]; polygon(a,b); //alternate
polygon(a,[b0,b1,b2,b3,b4]);
```

convexity

The convexity parameter specifies the maximum number of front sides (back sides) a ray intersecting the object might penetrate. This parameter is needed only for correct display of the object in OpenCSG preview mode and has no effect on the polyhedron rendering.



This image shows a 2D shape with a convexity of 4, as the ray indicated in red crosses the 2D shape a maximum of 4 times. The convexity of a 3D shape would be determined in a similar way. Setting it to 10 should work fine for most cases.

Created with the Personal Edition of HelpNDoc: [Free HTML Help documentation generator](#)

Text

The `text` module creates text as a 2D geometric object, using fonts installed on the local system or provided as separate font file.

Parameters

text

String. The text to generate.

size

Decimal. The generated text has an ascent (height above the baseline) of approximately the given value. Default is 10. Different fonts can vary somewhat and may not fill the size specified exactly, typically they render slightly smaller.

font

String. The name of the font that should be used. This is not the name of the font file, but the logical font name (internally handled by the fontconfig library). This can also include a style parameter, see below. A list of installed fonts & styles can be obtained using the font list dialog (Help -> Font List).

halign

String. The horizontal alignment for the text. Possible values are "left", "center" and "right". Default is "left".

valign

String. The vertical alignment for the text. Possible values are "top", "center", "baseline" and "bottom". Default is "baseline".

spacing

Decimal. Factor to increase/decrease the character spacing. The default value of 1 results in the normal spacing for the font, giving a value greater than 1 causes the letters to be spaced further apart.

direction

String. Direction of the text flow. Possible values are "ltr" (left-to-right), "rtl" (right-to-left),

"ttb" (top-to-bottom) and "btt" (bottom-to-top). Default is "ltr".

language

String. The language of the text. Default is "en".

script

String. The script of the text. Default is "latin".

\$fn

used for subdividing the curved path segments provided by freetype

Example

```
text ( "OpenSCAD" );
```

Note

To allow specification of particular [Unicode](#) characters you can specify them in a string with the following escape codes;

\x03 - single hex character (only allowed values are 01h - 7fh)

\u0123 - unicode char with 4 hexadecimal digits (note: Lowercase)

\U012345 - unicode char with 6 hexadecimal digits (note: Uppercase)

Example

```
t="\u20AC10 \u263A"; // 10 euro and a smileie
```



Example 1: Result.

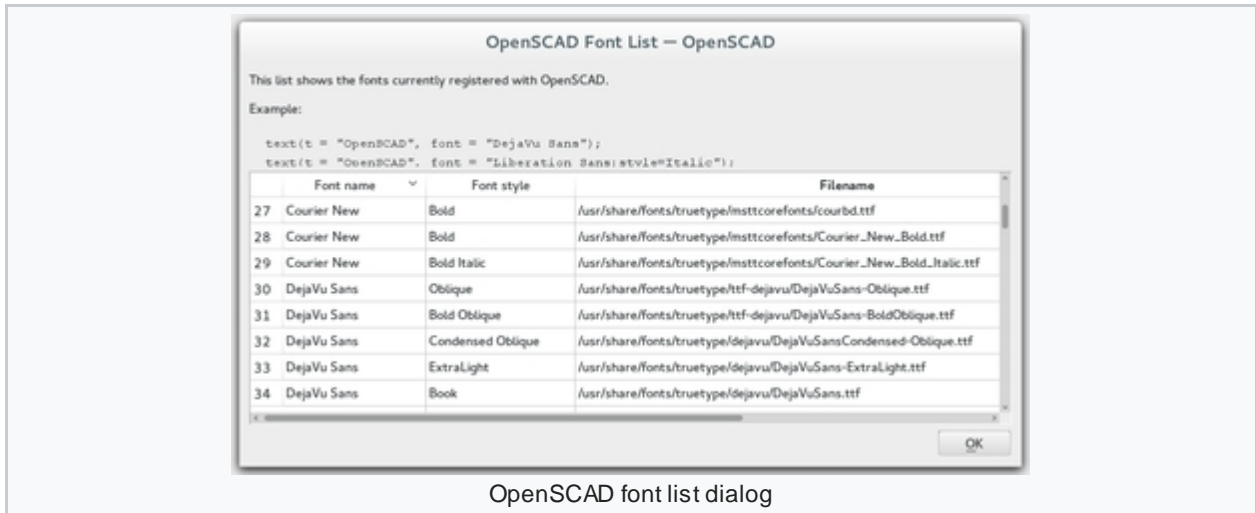
Created with the Personal Edition of HelpNDoc: [Full-featured EPub generator](#)

Using Fonts & Styles

Fonts are specified by their logical font name; in addition a style parameter can be added to select a specific font style like "**bold**" or "*italic*", such as:

```
font="Liberation Sans:style=Bold Italic"
```

The font list dialog (available under Help > Font List) shows the font name and the font style for each available font. For reference, the dialog also displays the location of the font file. You can drag a font in the font list, into the editor window to use in the text() statement.



OpenSCAD includes the fonts *Liberation Mono*, *Liberation Sans*, *Liberation Sans Narrow* and *Liberation Serif*. Hence, as fonts in general differ by platform type, use of these included fonts is likely to be portable across platforms.

For common/casual text usage, the specification of one of these fonts is **recommended** for this reason. Liberation Sans is the default font to encourage this.

In addition to the installed fonts, it's possible to add project specific font files. Supported font file formats are **TrueType** Fonts (*.ttf) and **OpenType** Fonts (*.otf). The files need to be registered with `use<>`.

```
use <ttf/paratype-serif/PTF55F.ttf>
```

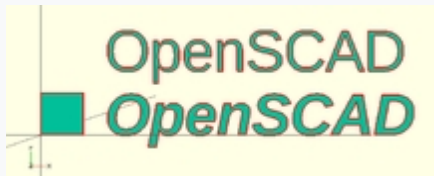
After the registration, the font is listed in the font list dialog, so in case logical name of a font is unknown, it can be looked up as it was registered.

OpenSCAD uses fontconfig to find and manage fonts, so it's possible to list the system configured fonts on command line using the fontconfig tools in a format similar to the GUI dialog.

```
$ fc-list -f "%-60{%{family[0]}%{:style[0]=}}%{file}\n" | sort ...
Liberation Mono:style=Bold
Italic /usr/share/fonts/truetype/liberation2/LiberationMono-BoldItalic.ttf
Liberation
Mono:style=Bold /usr/share/fonts/truetype/liberation2/LiberationMono-
Bold.ttf Liberation
Mono:style=Italic /usr/share/fonts/truetype/liberation2/LiberationMono-
Italic.ttf Liberation
Mono:style=Regular /usr/share/fonts/truetype/liberation2/LiberationMono-
Regular.ttf ...
```

Example

```
square(10); translate([15, 15]) { text("OpenSCAD", font =
"Liberation Sans"); } translate([15, 0]) { text("OpenSCAD", font =
"Liberation Sans:style=Bold Italic"); }
```



Example 2: Result.

Created with the Personal Edition of HelpNDoc: [Free EPub producer](#)

Vertical Alignment

top

The text is aligned with the top of the bounding box at the given Y coordinate.

center

The text is aligned with the center of the bounding box at the given Y coordinate.

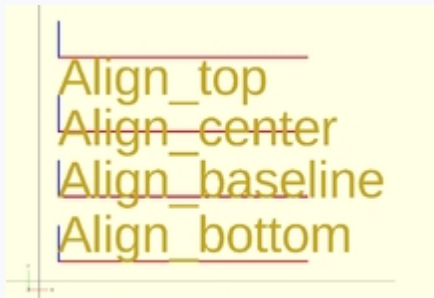
baseline

The text is aligned with the font baseline at the given Y coordinate. This is the default.

bottom

The text is aligned with the bottom of the bounding box at the given Y coordinate.

```
text = "Align"; font = "Liberation Sans"; valign = [ [ 0, "top"],
[ 40, "center"], [ 75, "baseline"], [110, "bottom"] ]; for (a =
valign) { translate([10, 120 - a[0], 0]) { color("red") cube([135,
1, 0.1]); color("blue") cube([1, 20, 0.1]); linear_extrude(height
= 0.5) { text(text = str(text,"_",a[1]), font = font, size = 20,
valign = a[1]); } } }
```



OpenSCAD vertical text alignment

Created with the Personal Edition of HelpNDoc: [What is a Help Authoring tool?](#)

Horizontal Alignment

left

The text is aligned with the left side of the bounding box at the given X coordinate. This is the default.

center

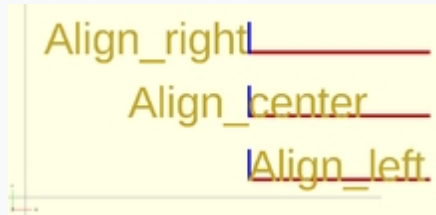
The text is aligned with the center of the bounding box at the given X coordinate.

right

The text is aligned with the right of the bounding box at the given X coordinate.

```
text = "Align"; font = "Liberation Sans"; halign = [ [10, "left"],
[50, "center"], [90, "right"] ]; for (a = halign)
{ translate([140, a[0], 0]) { color("red") cube([115, 2,0.1]);
color("blue") cube([2, 20,0.1]); linear_extrude(height = 0.5)
```

```
{ text(text = str(text,"_",a[1]), font = font, size = 20, halign = a[1]); } }
```



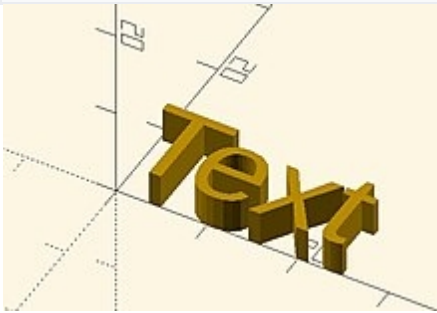
OpenSCAD horizontal text alignment

Created with the Personal Edition of HelpNDoc: [Full-featured multi-format Help generator](#)

3D Text

Text can be changed from a 2 dimensional object into a 3D object by using the [linear_extrude](#) function.

```
//3d Text Example linear_extrude(4) text("Text");
```



Created with the Personal Edition of HelpNDoc: [Easy CHM and documentation editor](#)

2D to 3D Projection

Extrusion is the process of creating an object with a fixed cross-sectional profile. OpenSCAD provides two commands to create 3D solids from a 2D shape: `linear_extrude()` and `rotate_extrude()`. Linear extrusion is similar to pushing Playdoh through a press with a die of a specific shape.

Rotational extrusion is similar to the process of **turning** or "throwing" a bowl on the **Potter's wheel**.

Both extrusion methods work on a (possibly disjointed) 2D shape which exists on the X-Y plane. While transformations that operates on both 2D shapes and 3D solids can move a shape off the X-Y plane, when the extrusion is performed the end result is not very intuitive. What actually happens is that any information in the third coordinate (the Z coordinate) is ignored for any 2D shape, this process amounts to an implicit [projection\(\)](#) performed on any 2D shape before the extrusion is executed. It is recommended to perform extrusion on shapes that remains strictly on the X-Y plane.



`linear_extrude()` works like a Playdoh extrusion press



`rotate_extrude()` emulates throwing a vessel

Created with the Personal Edition of HelpNDoc: [Generate EPub eBooks with ease](#)

Linear Extrude

Linear Extrusion is a operation that takes a 2D object as input and generates a 3D object as a result.

In OpenSCAD Extrusion is always performed on the projection (shadow) of the 2d object xy plane and along the **Z** axis; so if you rotate or apply other transformations to the 2d object before extrusion, it's shadow shape is what is extruded.

Although the extrusion is linear along the **Z** axis, a twist parameter is available that causes the object to be rotated around the **Z** axis as it is extruding upward. This can be used to rotate the object at it's center, as if it is a spiral pillar, or produce a helical extrusion around the **Z** axis, like a pig's tail.

A scale parameter is also included so that the object can be expanded or contracted over the extent of the extrusion, allowing extrusions to be flared inward or outward



Created with the Personal Edition of HelpNDoc: [Full-featured EPub generator](#)

Usage

```
linear_extrude(height = fanwidth, center = true, convexity = 10,  
twist = -fanrot, slices = 20, scale = 1.0, $fn = 16) {...}
```

You must use parameter names due to a backward compatibility issue.

`height` must be positive.

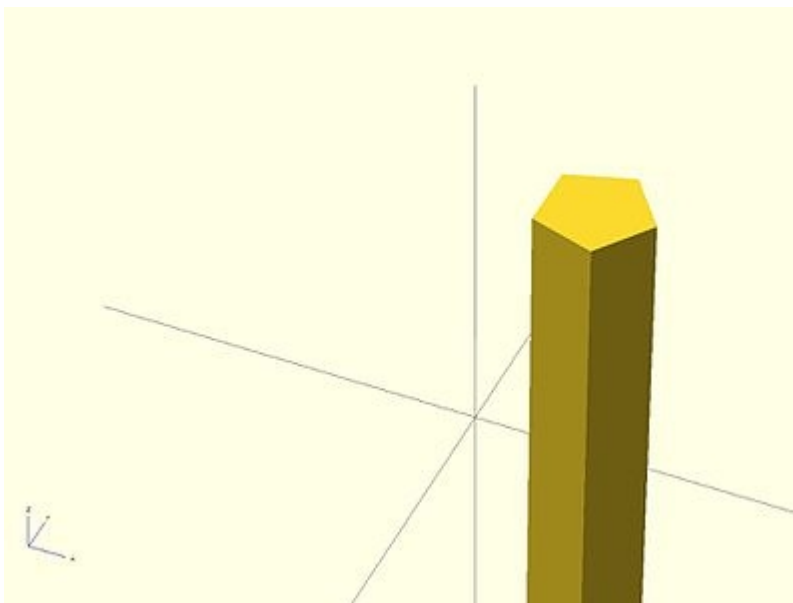
`$fn` is optional and specifies the resolution of the `linear_extrude` (higher number brings more "smoothness", but more computation time is needed).

If the extrusion fails for a non-trivial 2D shape, try setting the `convexity` parameter (the default is not 10, but 10 is a "good" value to try). See explanation further down.

Created with the Personal Edition of HelpNDoc: [Free EPub producer](#)

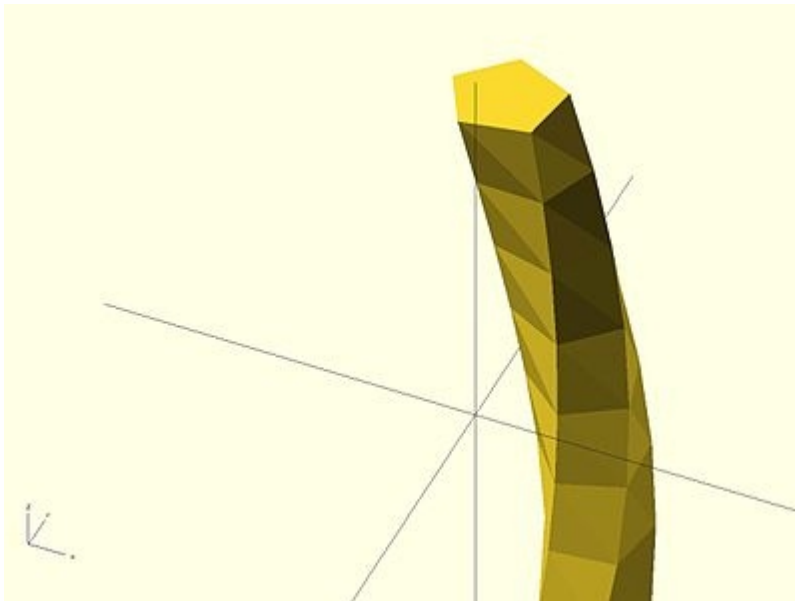
Twist

Twist is the number of degrees of through which the shape is extruded. Setting the parameter `twist = 360` extrudes through one revolution. The twist direction follows the left hand rule.

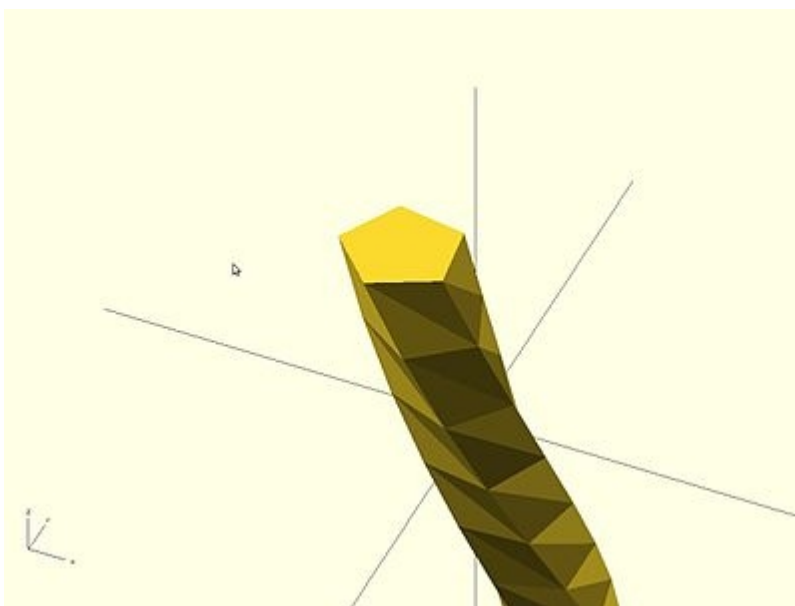


0° of Twist

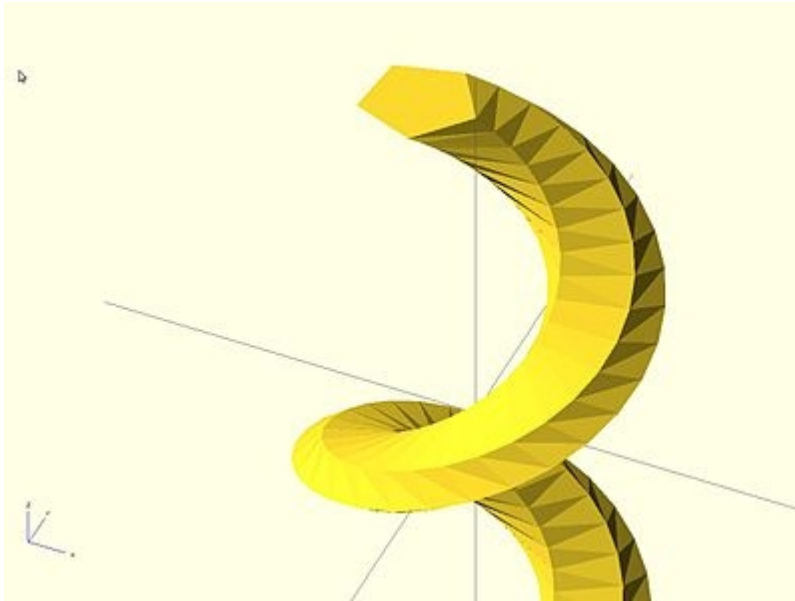
```
linear_extrude(height = 10, center = true, convexity = 10, twist =  
0) translate([2, 0, 0]) circle(r = 1);
```

**-100° of Twist**

```
linear_extrude(height = 10, center = true, convexity = 10, twist =  
-100) translate([2, 0, 0]) circle(r = 1);
```

**100° of Twist**

```
linear_extrude(height = 10, center = true, convexity = 10, twist =  
100) translate([2, 0, 0]) circle(r = 1);
```



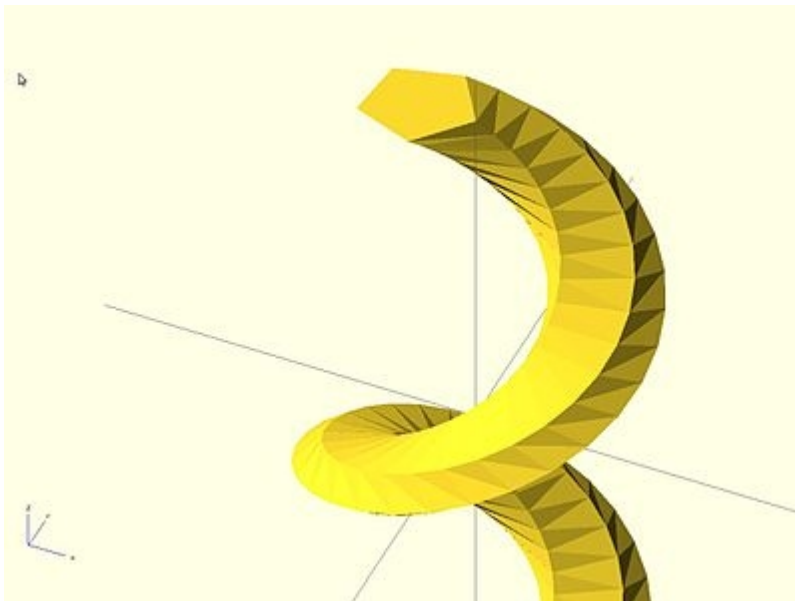
-500° of Twist

```
linear_extrude(height = 10, center = true, convexity = 10, twist =  
-500) translate([2, 0, 0]) circle(r = 1);
```

Created with the Personal Edition of HelpNDoc: [Create help files for the Qt Help Framework](#)

Center

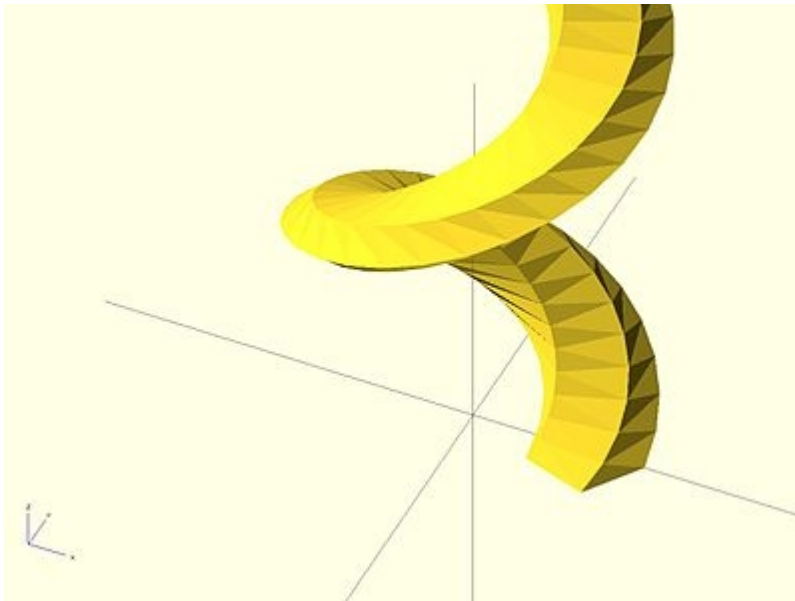
It is similar to the parameter center of cylinders. If `center` is false the linear extrusion Z range is from 0 to height; if it is true, the range is from -height/2 to height/2.



center = true

```
linear_extrude(height = 10, center = true, convexity = 10, twist =
```

```
-500) translate([2, 0, 0]) circle(r = 1);
```

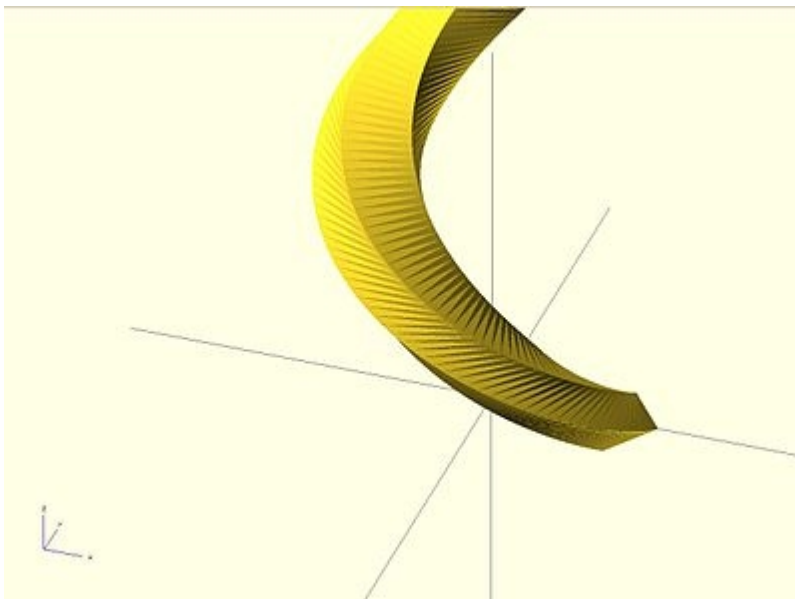
**center = false**

```
linear_extrude(height = 10, center = false, convexity = 10, twist  
= -500) translate([2, 0, 0]) circle(r = 1);
```

Created with the Personal Edition of HelpNDoc: [Free Web Help generator](#)

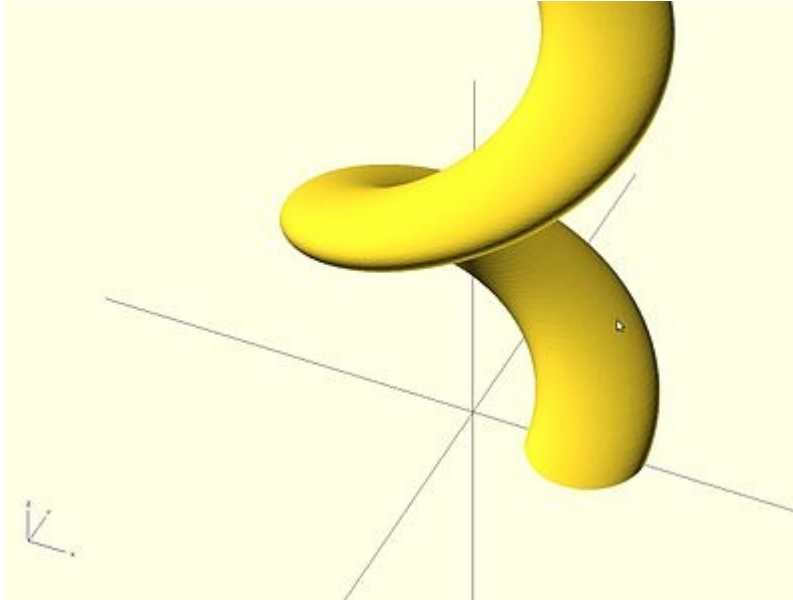
Mesh Refinement

The slices parameter defines the number of intermediate points along the Z axis of the extrusion. Its default increases with the value of twist. Explicitly setting slices may improve the output refinement.



```
linear_extrude(height = 10, center = false, convexity = 10, twist
= 360, slices = 100) translate([2, 0, 0]) circle(r = 1);
```

The [special variables](#) \$fn, \$fs and \$fa can also be used to improve the output. If slices is not defined, its value is taken from the defined \$fn value.



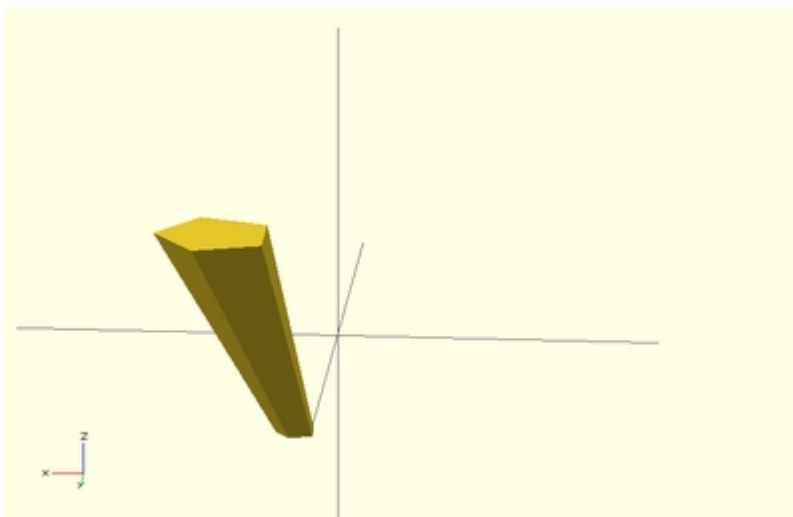
```
linear_extrude(height = 10, center = false, convexity = 10, twist
= 360, $fn = 100) translate([2, 0, 0]) circle(r = 1);
```

Created with the Personal Edition of HelpNDoc: [Full-featured EPub generator](#)

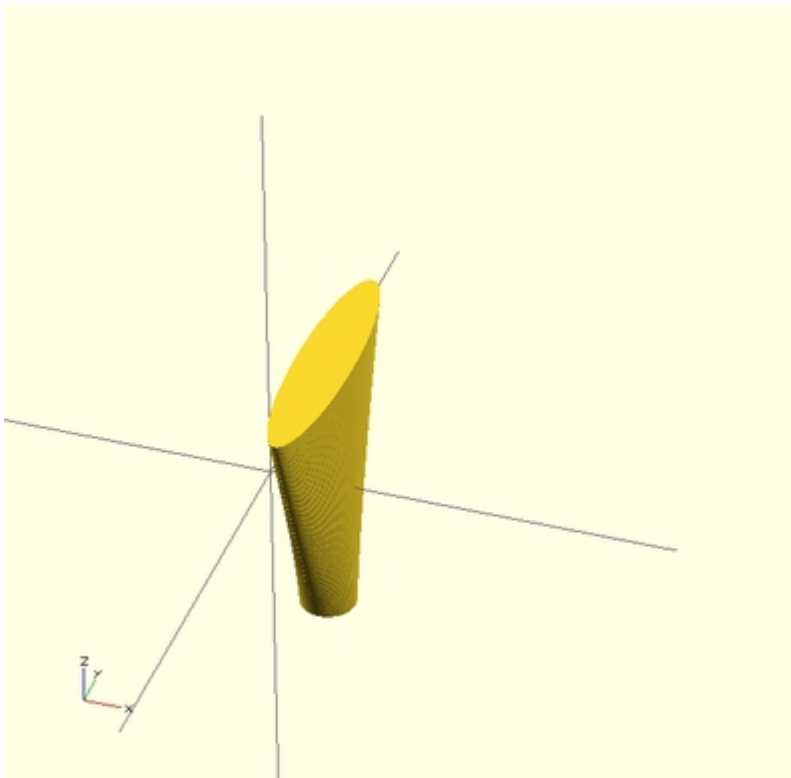
Scale

Scales the 2D shape by this value over the height of the extrusion. Scale can be a scalar or a vector:

```
linear_extrude(height = 10, center = true, convexity = 10,
scale=3) translate([2, 0, 0]) circle(r = 1);
```



```
linear_extrude(height = 10, center = true, convexity = 10,
scale=[1,5], $fn=100) translate([2, 0, 0]) circle(r = 1);
```



Note that if `scale` is a vector, the resulting side walls may be nonplanar. Use `twist=0` and the `slices` parameter to avoid [asymmetry](#).

```
linear_extrude(height=10, scale=[1,0.1], slices=20, twist=0)
polygon(points=[[0,0],[20,10],[20,-10]]);
```

Created with the Personal Edition of HelpNDoc: [Free iPhone documentation generator](#)

Rotate Extrude

Rotational extrusion spins a 2D shape around the Z-axis to form a solid which has rotational symmetry. One way to think of this operation is to imagine a Potter's wheel placed on the X-Y plane with its axis of rotation pointing up towards +Z. Then place the to-be-made object on this virtual Potter's wheel (possibly extended down below the X-Y plane towards -Z, take the cross-section of this object on the X-Z plane but keep only the right half ($X \geq 0$). That is the 2D shape that need to be fed to `rotate_extrude()` as the child in order to generate this solid.

Since a 2D shape is rendered by OpenSCAD on the X-Y plane, an alternative way to think of this operation is as follows: spins a 2D shape around the Y-axis to form a solid. The resultant solid is placed so that its axis of rotation lies along the Z-axis.

Just like the `linear_extrude`, the extrusion is always performed on the projection of the 2D polygon to the XY plane. Transformations like `rotate`, `translate`, etc. applied to the 2D polygon before extrusion modify the projection of the 2D polygon to the XY plane and therefore also modify the appearance of the final 3D object.

- A translation in Z of the 2D polygon has no effect on the result (as also the projection is not affected).

- A translation in X increases the diameter of the final object.
- A translation in Y results in a shift of the final object in Z direction.
- A rotation about the X or Y axis distorts the cross section of the final object, as also the projection to the XY plane is distorted.

Don't get confused, as OpenSCAD renders 2D polygons with a certain height in the Z direction, so the 2D object (with it's height) appears to have a bigger projection to the XY plane. But for the projection to the XY plane and also for the later extrusion only the base polygon without height is used.

It can not be used to produce a helix or screw threads.

The 2D shape **must** lie completely on either the right (recommended) or the left side of the Y-axis. More precisely speaking, **every** vertex of the shape must have either $x \geq 0$ or $x \leq 0$. If the shape spans the X axis a warning appears in the console windows and the `rotate_extrude()` is ignored. If the 2D shape touches the Y axis, i.e. at $x=0$, it **must** be a line that touches, not a point, as a point results in a zero thickness 3D object, which is invalid and results in a CGAL error. For OpenSCAD versions prior to 2016.xxxx, if the shape is in the negative axis the the resulting faces are oriented inside-out, which may cause undesired effects.

Created with the Personal Edition of HelpNDoc: [Create help files for the Qt Help Framework](#)

Usage

Usage[\[edit\]](#)

```
rotate_extrude(angle = 360, convexity = 2) {...}
```

You must use parameter names due to a backward compatibility issue.

convexity

If the extrusion fails for a non-trivial 2D shape, try setting the convexity parameter (the default is not 10, but 10 is a "good" value to try). See explanation further down.

angle [**Note:** Requires version **2019.05**]

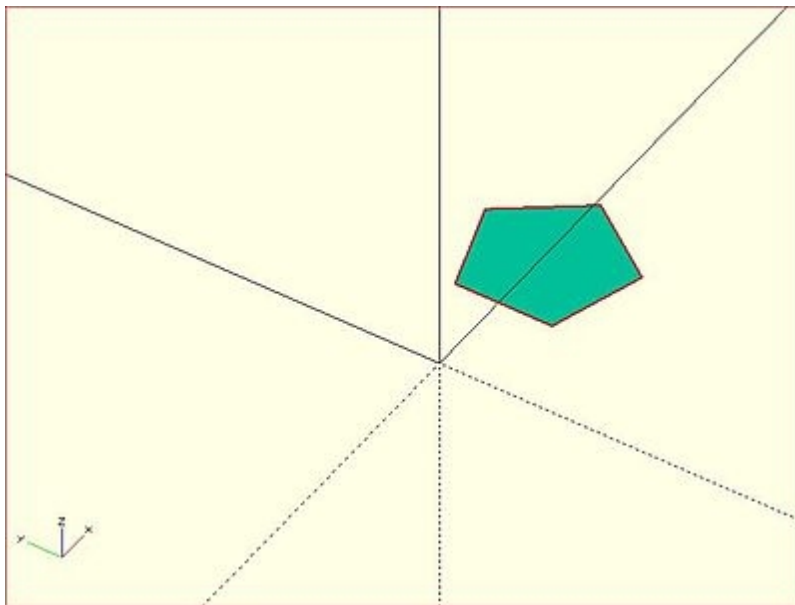
Defaults to 360. Specifies the number of degrees to sweep, starting at the positive X axis. The direction of the sweep follows the **Right Hand Rule**, hence a negative angle sweeps clockwise.



Right-hand grip rule

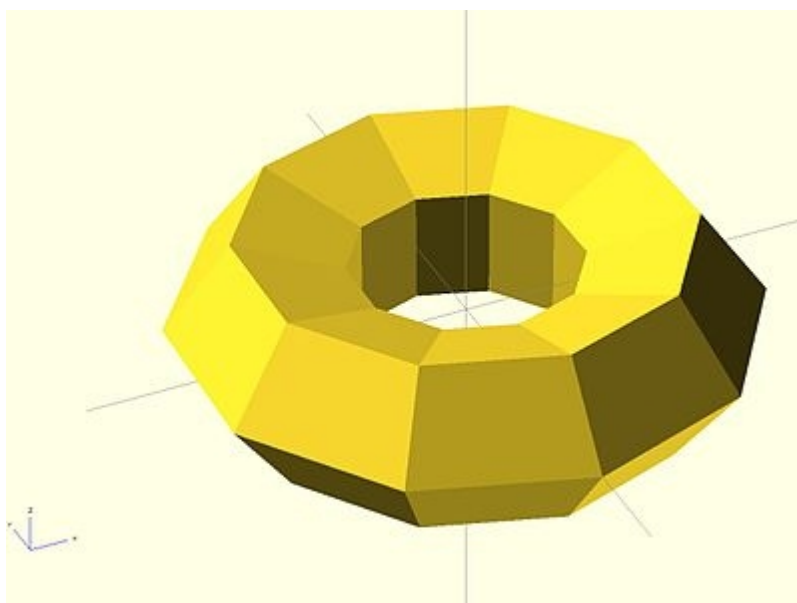
Created with the Personal Edition of HelpNDoc: [Create cross-platform Qt Help files](#)

Examples

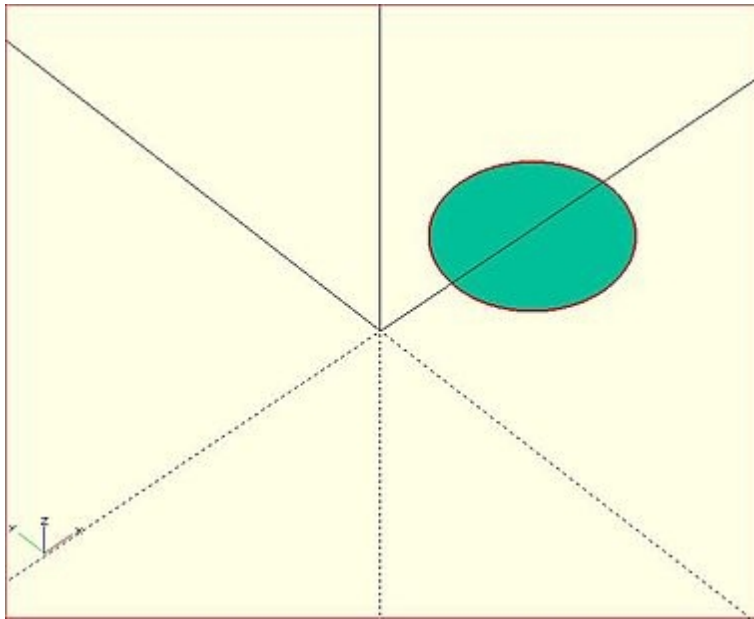


A simple torus can be constructed using a rotational extrude.

```
rotate_extrude(convexity = 10) translate([2, 0, 0]) circle(r = 1);
```

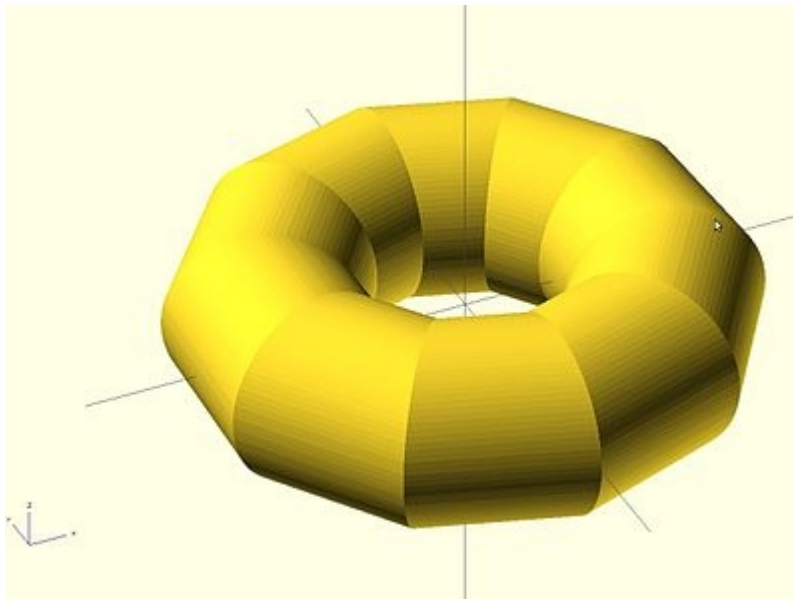


Mesh Refinement



Increasing the number of fragments composing the 2D shape improves the quality of the mesh, but takes longer to render.

```
rotate_extrude(convexity = 10) translate([2, 0, 0]) circle(r = 1,
$fn = 100);
```



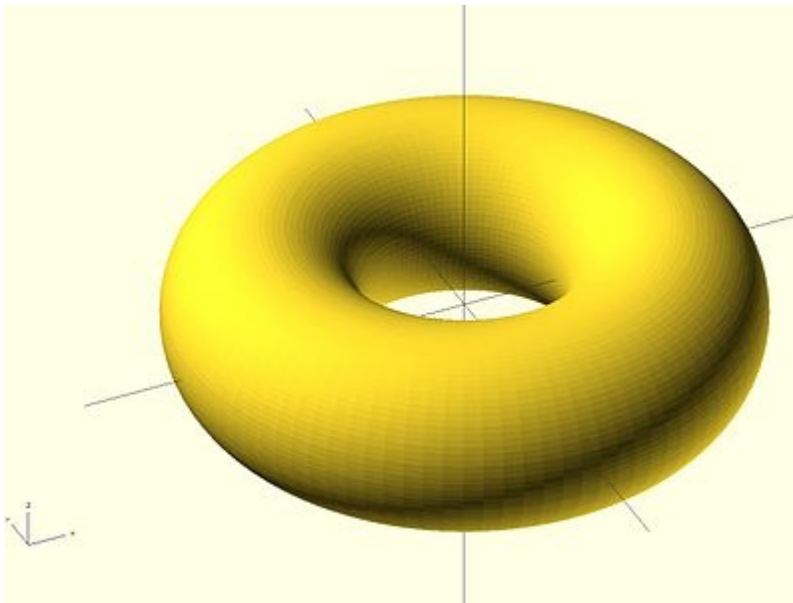
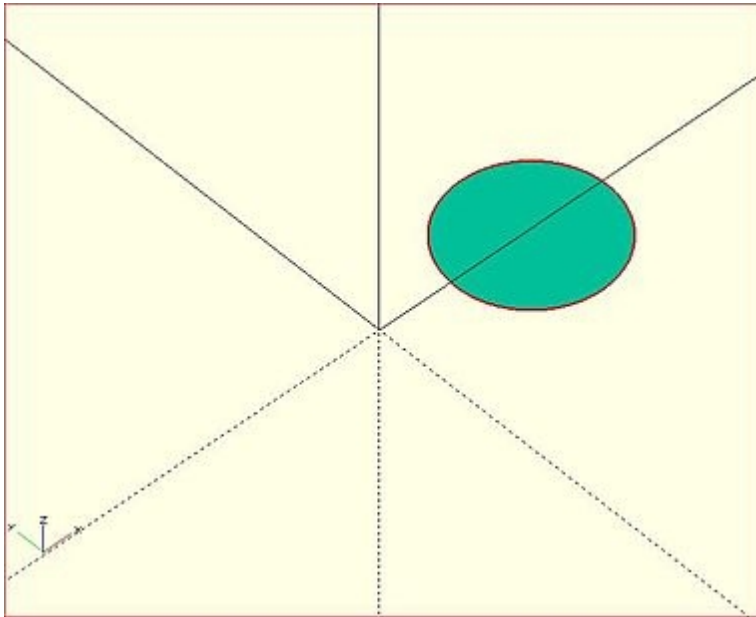
The number of fragments used by the extrusion can also be increased.

```
rotate_extrude(convexity = 10, $fn = 100) translate([2, 0, 0])
circle(r = 1, $fn = 100);
```

Using the parameter angle (with OpenSCAD versions 2016.xx), a hook can be modeled .

```
translate([0,60,0]) rotate_extrude(angle=270, convexity=10)
translate([40, 0]) circle(10); rotate_extrude(angle=90,
convexity=10) translate([20, 0]) circle(10); translate([20,0,0])
```

```
rotate([90,0,0]) cylinder(r=10,h=80);
```



OpenSCAD - a hook

Created with the Personal Edition of HelpNDoc: [Easily create PDF Help documents](#)

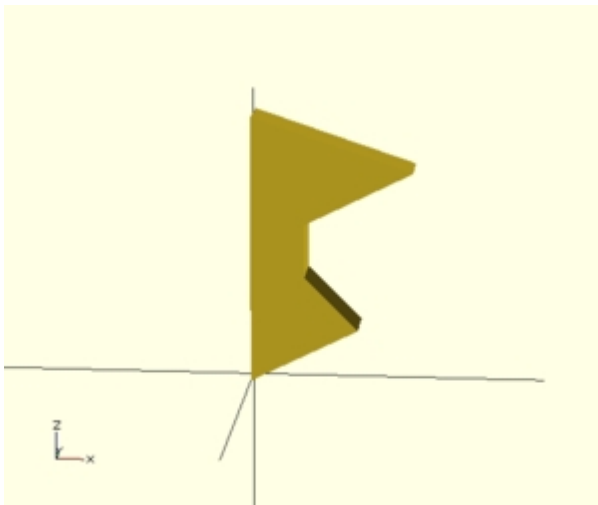
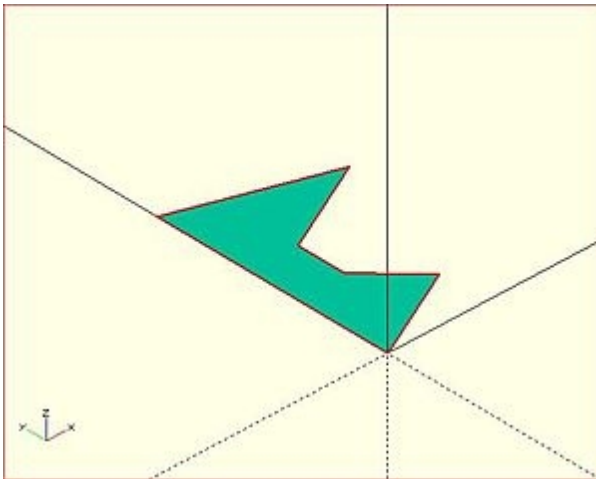
Extruding a Polygon

Extrusion can also be performed on polygons with points chosen by the user.

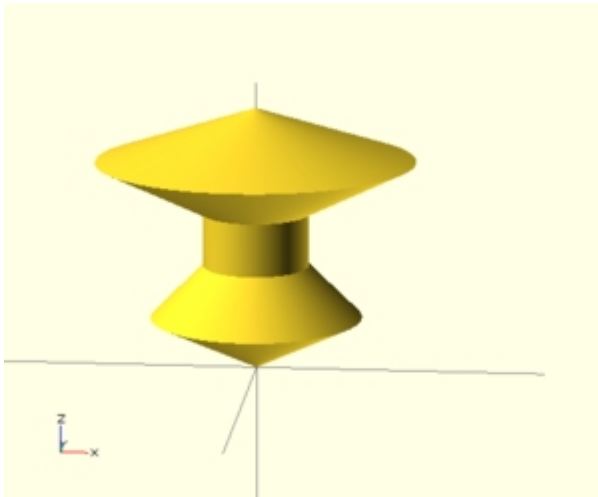
Here is a simple polygon and its 200 step rotational extrusion. (Note it has been rotated 90 degrees to show how the rotation appears; the `rotate_extrude()` needs it flat).

```
rotate([90,0,0]) polygon( points=[[0,0],[2,1],[1,2],[1,3],[3,4],  
[0,5]] );
```

```
rotate_extrude($fn=200) polygon( points=[[0,0],[2,1],[1,2],[1,3],  
[3,4],[0,5]] );
```



For more information on polygons, please see: [2D Primitives: Polygon](#).



Created with the Personal Edition of HelpNDoc: [Create cross-platform Qt Help files](#)

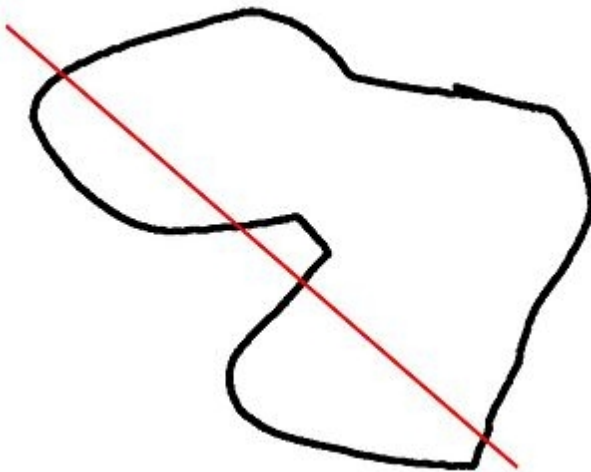
Description of extrude Parameters

Extrude parameters for all extrusion modes[\[edit\]](#)

convexity

Integer. The convexity parameter specifies the maximum number of front sides (back sides) a ray intersecting the object might penetrate.

This parameter is only needed for correctly displaying the object in OpenCSG preview mode and has no effect on the polyhedron rendering.



This image shows a 2D shape with a convexity of 4, as the ray indicated in red crosses the 2D shape a maximum of 4 times. The convexity of a 3D shape would be determined in a similar way. Setting it to 10 should work fine for most cases.

Extrude parameters for linear extrusion only[\[edit\]](#)

height	The extrusion height
center	If true, the solid is centered after extrusion
twist	The extrusion twist in degrees
slices	Similar to special variable \$fn without being passed down to the child 2D shape.
scale	Scales the 2D shape by this value over the height of the extrusion.

Created with the Personal Edition of HelpNDoc: [iPhone web sites made easy](#)

Transforms

Basic concept

Transformation affect the child nodes and as the name implies transforms them in various ways such as moving/rotating or scaling the child. Cascading transformations are used to apply a variety of transforms to a final child. Cascading is achieved by nesting statements i.e.

```
rotate([45,45,45]) translate([10,20,30]) cube(10);
```

Transformations can be applied to a group of child nodes by using '{' and '}' to enclose the subtree e.g.

```
translate([0,0,-5])
{
  cube(10);
  cylinder(r=5,h=10);
}
```

Transformations are written before the object they affect.

Imagine commands like translate, mirror and scale as verbs. Commands like color are like adjectives that describe the object.

Notice that there is no semicolon following transformation command.

Advanced concept

As OpenSCAD uses different libraries to implement capabilities this can introduce some inconsistencies to the F5 preview behavior of transformations. Traditional transforms (translate, rotate, scale, mirror & multimatrix) are performed using OpenGL in preview, while other more advanced transforms, such as resize, perform a CGAL operation, behaving like a CSG operation affecting the underlying object, not just transforming it. In particular this can affect the display of modifier characters, specifically "#" and "%", where the highlight may not display intuitively, such as highlighting the pre-resized object, but highlighting the post-scaled object.

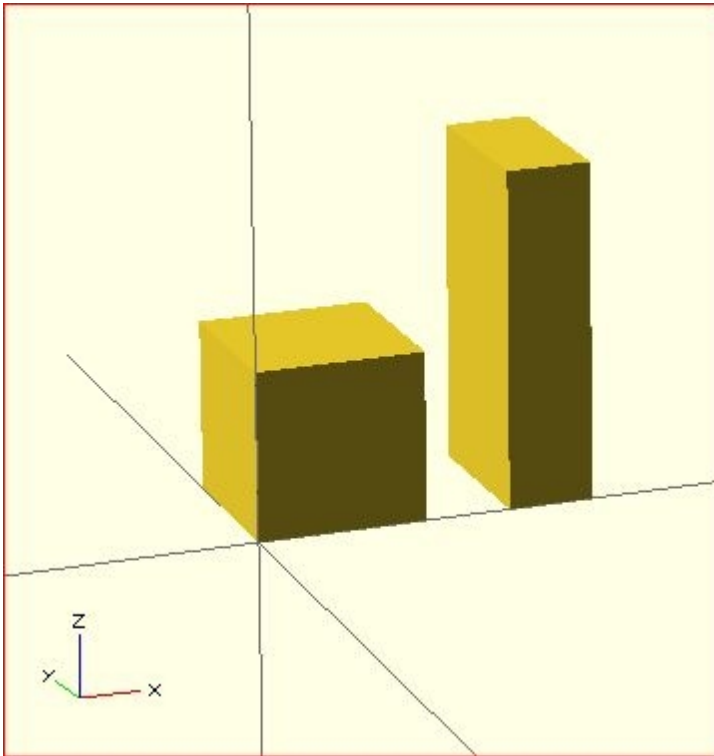
Created with the Personal Edition of HelpNDoc: [Create help files for the Qt Help Framework](#)

scale

Scales its child elements using the specified vector. The argument name is optional.

Usage Example: `scale(v = [x, y, z]) { ... }`

```
cube(10);  
translate([15,0,0]) scale([0.5,1,2]) cube(10);
```



Created with the Personal Edition of HelpNDoc: [Produce electronic books easily](#)

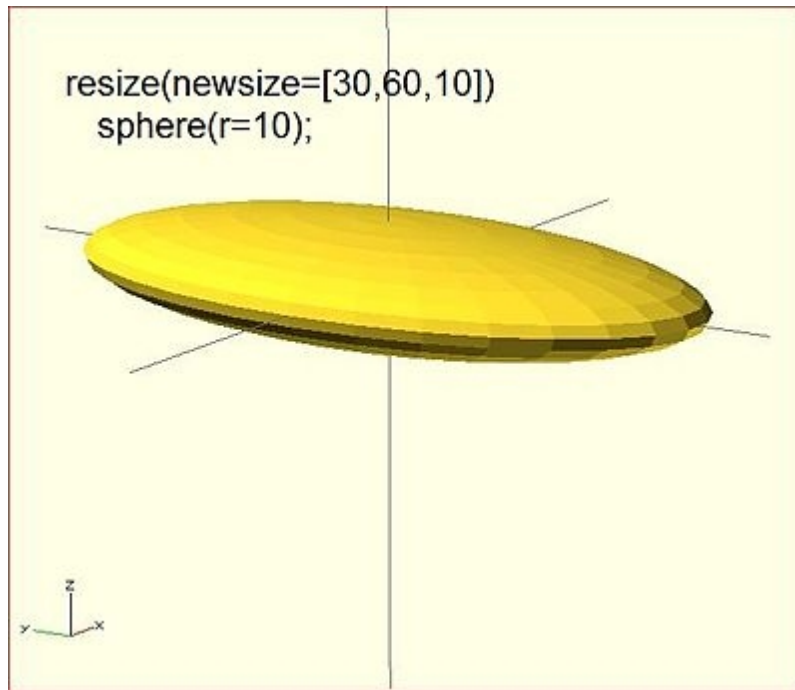
resize

Modifies the size of the child object to match the given x,y, and z.

`resize()` is a CGAL operation, and like others such as `render()` operates with full geometry, so even in preview this takes time to process.

Usage Example:

```
// resize the sphere to extend 30 in x, 60 in y, and 10 in the z  
directions.  
resize(newsize=[30,60,10]) sphere(r=10);
```



If x,y, or z is 0 then that dimension is left as-is.

```
// resize the 1x1x1 cube to 2x2x1
resize([2,2,0]) cube();
```

If the 'auto' parameter is set to true, it auto-scales any 0-dimensions to match. For example.

```
// resize the 1x2x0.5 cube to 7x14x3.5
resize([7,0,0], auto=true) cube([1,2,0.5]);
```

The 'auto' parameter can also be used if you only wish to auto-scale a single dimension, and leave the other as-is.

```
// resize to 10x8x1. Note that the z dimension is left alone.
resize([10,0,0], auto=[true,true,false]) cube([5,4,1]);
```

Created with the Personal Edition of HelpNDoc: [What is a Help Authoring tool?](#)

rotate

Rotates its child 'a' degrees about the axis of the coordinate system or around an arbitrary axis. The argument names are optional if the arguments are given in the same order as specified.

```
//Usage:
rotate(a = deg_a, v = [x, y, z]) { ... } // or
rotate(deg_a, [x, y, z]) { ... }
rotate(a = [deg_x, deg_y, deg_z]) { ... }
rotate([deg_x, deg_y, deg_z]) { ... }
```

The 'a' argument (deg_a) can be an array, as expressed in the later usage above; when deg_a is an array, the 'v' argument is ignored. Where 'a' specifies *multiple axes* then the rotation is applied in the following order: x, y, z. That means the code:

```
rotate(a=[ax,ay,az]) { ... }
```

is equivalent to:

```
rotate(a=[0,0,az]) rotate(a=[0,ay,0]) rotate(a=[ax,0,0]) { ... }
```

The optional argument 'v' is a vector and allows you to set an arbitrary axis about which the object is rotated.

For example, to flip an object upside-down, you can rotate your object 180 degrees around the 'y' axis.

```
rotate(a=[0,180,0]) { ... }
```

This is frequently simplified to

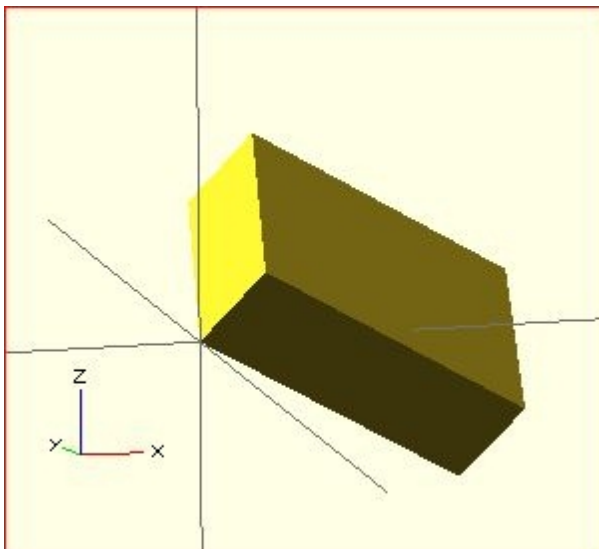
```
rotate([0,180,0]) { ... }
```

When specifying a single axis the 'v' argument allows you to specify which axis is the basis for rotation. For example, the equivalent to the above, to rotate just around y

```
rotate(a=180, v=[0,1,0]) { ... }
```

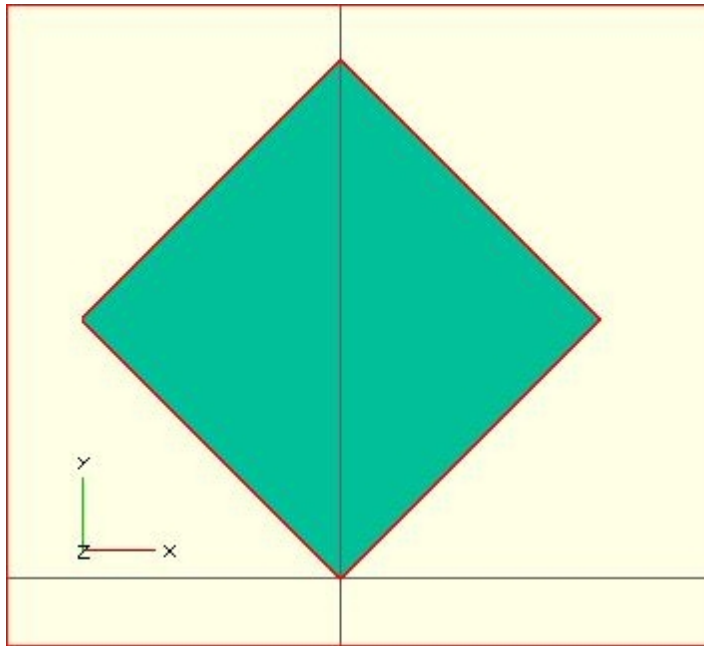
When specifying a single axis, 'v' is a **vector** defining an arbitrary axis for rotation; this is different from the *multiple axis* above. For example, rotate your object 45 degrees around the axis defined by the vector [1,1,0],

```
rotate(a=45, v=[1,1,0]) { ... }
```



Rotate with a *single scalar argument* rotates around the Z axis. This is useful in 2D contexts where that is the only axis for rotation. For example:

```
rotate(45) square(10);
```

Created with the Personal Edition of HelpNDoc: [Produce electronic books easily](#)

Rotation rule Help

For the case of:

```
rotate([a, b, c]) { ... };
```

"a" is a rotation about the X axis, from the +Y axis, toward the +Z axis.

"b" is a rotation about the Y axis, from the +Z axis, toward the +X axis.

"c" is a rotation about the Z axis, from the +X axis, toward the +Y axis.

These are all cases of the **Right Hand Rule**. Point your right thumb along the positive axis, your fingers show the direction of rotation.



Thus if "a" is fixed to zero, and "b" and "c" are manipulated appropriately, this is the *spherical coordinate system*.

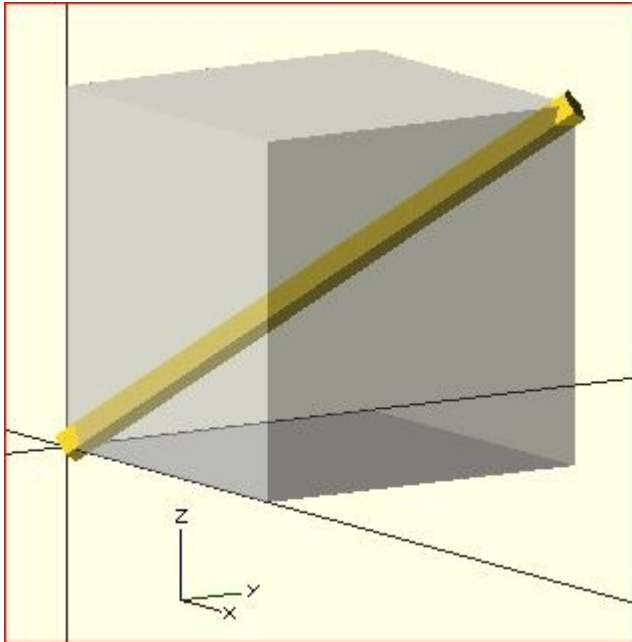
So, to construct a cylinder from the origin to some other point (x,y,z):

```
x= 10; y = 10; z = 10; // point coordinates of end of cylinder
length = norm([x,y,z]); // radial distance
```

```

b = acos(z/length); // inclination angle
c = atan2(y,x); // azimuthal angle
rotate([0, b, c]) cylinder(h=length, r=0.5);
%cube([x,y,z]); // corner of cube should coincide with end of cylinder

```



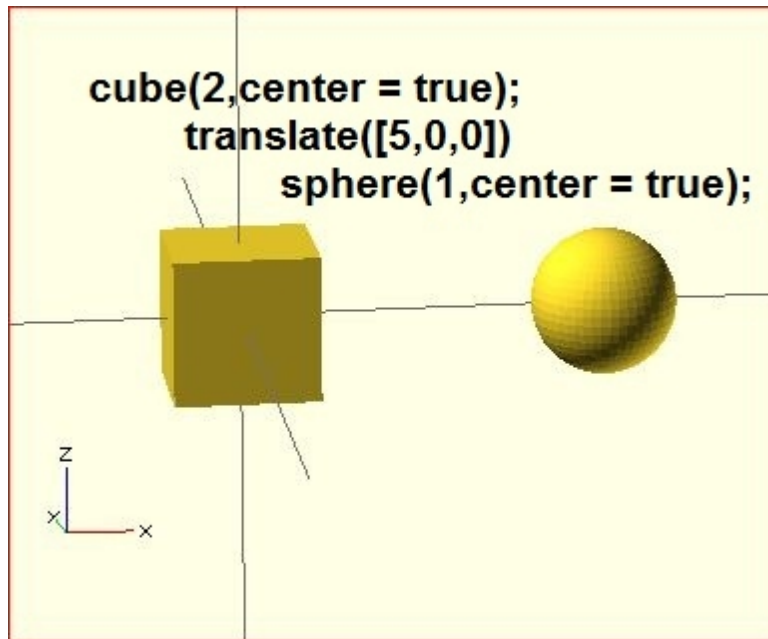
Created with the Personal Edition of HelpNDoc: [Easily create Help documents](#)

translate

Translates (moves) its child elements along the specified vector. The argument name is optional.

Example: `translate(v = [x, y, z]) { ... }`

```
cube(2,center = true); translate([5,0,0]) sphere(1,center = true);
```



Created with the Personal Edition of HelpNDoc: [Create help files for the Qt Help Framework](#)

mirror

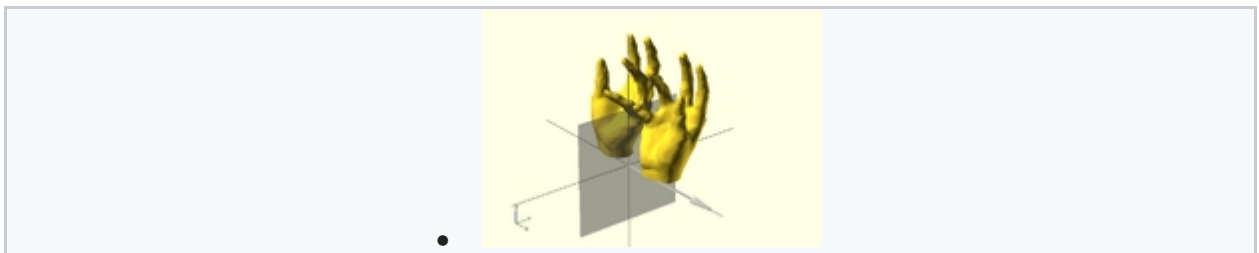
Mirrors the child element on a plane through the origin. The argument to `mirror()` is the normal vector of a plane intersecting the origin through which to mirror the object.

Function signature: [\[edit\]](#)

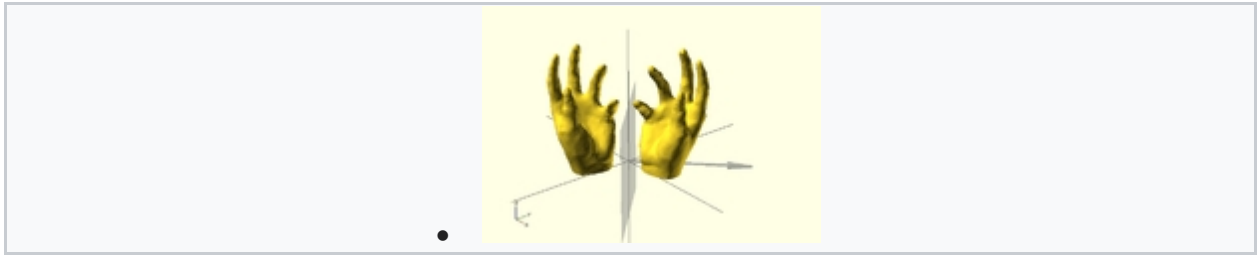
```
mirror(v= [x, y, z] ) { ... }
```

Examples [\[edit\]](#)

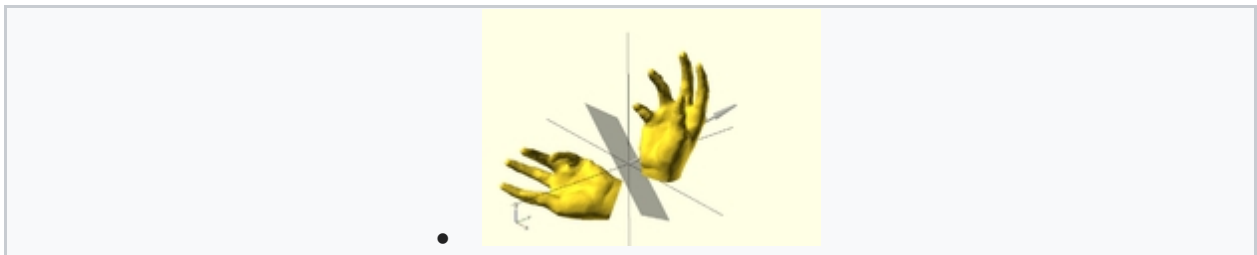
The original is on the right side. Note that `mirror` doesn't make a copy. Like `rotate` and `scale`, it changes the object.



```
hand(); // original
mirror([1,0,0]) hand();
```

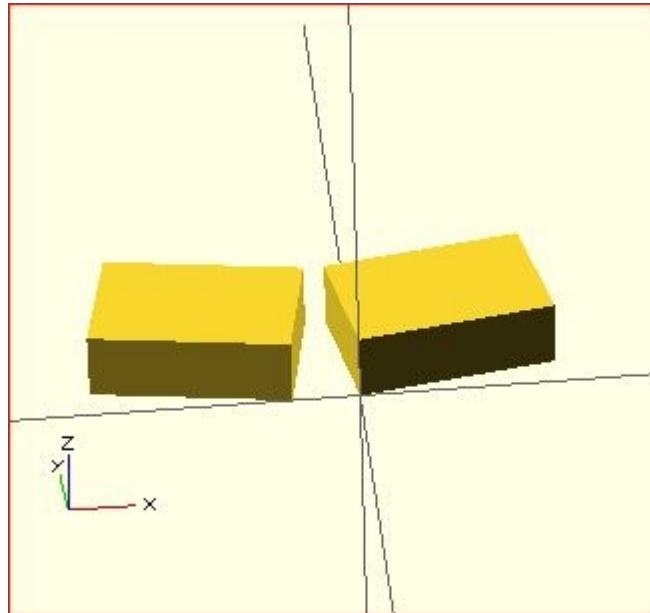


```
hand(); // original
mirror([1,1,0]) hand();
```



```
hand(); // original
mirror([1,1,1]) hand();
```

```
rotate([0,0,10]) cube([3,2,1]); mirror([1,0,0]) translate([1,0,0])
rotate([0,0,10]) cube([3,2,1]);
```



Created with the Personal Edition of HelpNDoc: [Full-featured Help generator](#)

multimatrix

Multiplies the geometry of all child elements with the given [affine transformation](#) matrix, where the matrix is 4×3 - a vector of 3 row vectors with 4 elements each, or a 4×4 matrix with the 4th row always forced to [0,0,0,1].

Usage: `multmatrix(m = [...]) { ... }`

This is a breakdown of what you can do with the independent elements in the matrix (for the first three rows):

[Scale X]	[Shear X along Y]	[Shear X along Z]	[Translate X]
[Shear Y along X]	[Scale Y]	[Shear Y along Z]	[Translate Y]
[Shear Z along X]	[Shear Z along Y]	[Scale Z]	[Translate Z]

The fourth row is forced to [0,0,0,1] and can be omitted unless you are combining matrices before passing to `multmatrix`, as it is not processed in OpenSCAD. Each matrix operates on the points of the given geometry as if each vertex is a 4 element vector consisting of a 3D vector with an implicit 1 as its 4th element, such as $v=[x, y, z, 1]$. The role of the implicit fourth row of m is to preserve the implicit 1 in the 4th element of the vectors, permitting the translations to work. The operation of `multmatrix` therefore performs $m*v$ for each vertex v . Any elements (other than the 4th row) not specified in m are treated as zeros.

This example rotates by 45 degrees in the XY plane and translates by [10,20,30], i.e. the same as `translate([10,20,30]) rotate([0,0,45])` would do.

```
angle=45;
multmatrix(m = [ [cos(angle), -sin(angle), 0, 10],
[ sin(angle), cos(angle), 0, 20],
[ 0, 0, 1, 30],
[ 0, 0, 0, 1]
]) union() {
cylinder(r=10.0,h=10,center=false);
cube(size=[10,10,10],center=false);
}
```

The following example demonstrates combining affine transformation matrices by matrix multiplication, producing in the final version a transformation equivalent to `rotate([0, -35, 0]) translate([40, 0, 0]) Obj();`. Note that the signs on the sin function appear to be in a different order than the above example, because the positive one must be ordered as x into y, y into z, z into x for the rotation angles to correspond to rotation about the other axis in a right-handed coordinate system.

```
y_ang=-35;
mrot_y = [ [ cos(y_ang), 0, sin(y_ang), 0],
[ 0, 1, 0, 0],
[ -sin(y_ang), 0, cos(y_ang), 0],
[ 0, 0, 0, 1]
];
mtrans_x = [ [1, 0, 0, 40],
[0, 1, 0, 0],
[0, 0, 1, 0],
[0, 0, 0, 1]
];
module Obj() {
cylinder(r=10.0,h=10,center=false);
cube(size=[10,10,10],center=false);
}
echo(mrot_y*mtrans_x);
Obj();
multmatrix(mtrans_x) Obj();
multmatrix(mrot_y * mtrans_x) Obj();
```

This example skews a model, which is not possible with the other transformations.

```
M = [ [ 1 , 0 , 0 , 0 ],
      [ 0 , 1 , 0.7, 0 ], // The "0.7" is the skew value; pushed along the y
      [ 0 , 0 , 1 , 0 ],  axis as z changes.
      [ 0 , 0 , 0 , 1 ] ] ;
multmatrix(M) { union() {
  cylinder(r=10.0,h=10,center=false);
  cube(size=[10,10,10],center=false); } }
```

More?[\[edit\]](#)

Learn more about it here:

- [Affine Transformations](#) on wikipedia
- <http://www.senocular.com/flash/tutorials/transformmatrix/>

Created with the Personal Edition of HelpNDoc: [Easily create EBooks](#)

color

Displays the child elements using the specified RGB color + alpha value. This is only used for the F5 preview as CGAL and STL (F6) do not currently support color. The alpha value defaults to 1.0 (opaque) if not specified.

Function signature:[\[edit\]](#)

```
color( c = [r, g, b, a] ) { ... } color( c = [r, g, b], alpha =
1.0 ) { ... } color( "#hexvalue" ) { ... } color( "colorname", 1.0
) { ... }
```

Note that the `r`, `g`, `b`, `a` values are limited to floating point values in the range **[0,1]** rather than the more traditional integers { 0 ... 255 }. However, nothing prevents you to using `R`, `G`, `B` values from {0 ... 255} with appropriate scaling: `color([R/255, G/255, B/255]) { ... }`

[Note: Requires version 2011.12] Colors can also be defined by name (case insensitive). For example, to create a red sphere, you can write `color("red") sphere(5);`. Alpha is specified as an extra parameter for named colors: `color("Blue",0.5) cube(5);`

[Note: Requires version 2019.05] Hex values can be given in 4 formats, `#rgb`, `#rgba`, `#rrggbb` and `#rrggbbaa`. If the alpha value is given in both the hex value and as separate alpha parameter, the alpha parameter takes precedence.

The available color names are taken from the World Wide Web consortium's [SVG color list](#). A chart of the color names is as follows,

(note that both spellings of grey/gray including slategrey/slategray etc are valid):

Purples	Blues	Greens	Yellows	Whites
Lavender	Aqua	GreenYellow	Gold	White
Thistle	Cyan	Chartreuse	Yellow	Snow
Plum	LightCyan	LawnGreen	LightYellow	Honeydew
Violet	PaleTurquoise	Lime	LemonChiffon	MintC
Orchid	Aquamarine	LimeGreen	LightGoldenrodYellow	
Fuchsia	Turquoise	PaleGreen		

Magenta	MediumTurquoise	LightGreen	PapayaWhip	ream
MediumOrchid	DarkTurquoise	MediumSpringGreen	Moccasin	Azure
MediumPurple	CadetBlue	SpringGreen	PeachPuff	Alice Blue
BlueViolet	SteelBlue	MediumSeaGreen	PaleGoldenrod	Blue
DarkViolet	LightSteelBlue	SeaGreen	Khaki	Ghost White
DarkOrchid	PowderBlue	ForestGreen	DarkKhaki	White Smoke
DarkMagenta	LightBlue	Green	Browns	White Smoke
Purple	SkyBlue	DarkGreen	Cornsilk	Seashell
Indigo	LightSkyBlue	YellowGreen	BlanchedAlmond	Beige
DarkSlateBlue	DeepSkyBlue	OliveDrab	Bisque	OldLace
SlateBlue	DodgerBlue	Olive	NavajoWhite	Floral White
MediumSlateBlue	CornflowerBlue	DarkOliveGreen	Wheat	Ivory
Pinks	RoyalBlue	MediumAquamarine	BurlyWood	AntiqueWhite
Pink	Blue	DarkSeaGreen	Tan	Linen
LightPink	MediumBlue	LightSeaGreen	RosyBrown	LavenderBlush
HotPink	DarkBlue	DarkCyan	SandyBrown	MistyRose
DeepPink	Navy	Teal	Goldenrod	Grays
MediumVioletRed	MidnightBlue	Oranges	DarkGoldenrod	Gainsboro
PaleVioletRed	Reds	LightSalmon	Peru	LightGrey
	IndianRed	Coral	Chocolate	Silver
	LightCoral	Tomato	SaddleBrown	DarkGray
	Salmon	OrangeRed	Sienna	Gray
	DarkSalmon	DarkOrange	Brown	DimGray
	LightSalmon	Orange	Maroon	LightSlateGray
	Red			SlateGray
	Crimson			DarkSlateGray
	FireBrick			Black
	DarkRed			

Example [\[edit\]](#)

Here's a code fragment that draws a wavy multicolor object

```
for(i=[0:36]) {
```

```

for(j=[0:36]) {
color( [0.5+sin(10*i)/2, 0.5+sin(10*j)/2, 0.5+sin(10*(i+j))/2] )
translate( [i, j, 0] )
cube( size = [1, 1, 11+10*cos(10*i)*sin(10*j)] );
}
}

```

↗ Being that $-1 \leq \sin(x) \leq 1$ then $0 \leq (1/2 + \sin(x)/2) \leq 1$, allowing for the RGB components assigned to color to remain within the [0,1] interval.

Chart based on "Web Colors" from Wikipedia

Example 2[\[edit\]](#)

In cases where you want to optionally set a color based on a parameter you can use the following trick:

```

module myModule(withColors=false) {
c=withColors?"red":undef;
color(c) circle(r=10);
}

```

Setting the colorname to undef keeps the default colors.



Created with the Personal Edition of HelpNDoc: [Full-featured EBook editor](#)

offset

Offset generates a new 2d interior or exterior outline from an existing outline. There are two modes of operation. radial and offset. The offset method creates a new outline who's sides are a fixed distance outer ($\delta > 0$) or inner ($\delta < 0$) from the original outline. The radial method creates a new outline as if a circle of some radius is rotated around the exterior ($r > 0$) or interior ($r < 0$) original outline.

The construction methods can either produce an outline that is interior or exterior to the original outline. For exterior outlines the corners can be given an optional chamfer.

- This transform is useful for making thin walls, by subtracting a negative-offset construction from the original, or the original from a Positive offset construction.
- Fillet: offset($r=-3$) offset($\delta=+3$) rounds all inside (concave) corners, and leaves flat walls unchanged. However, holes less than $2*r$ in diameter vanish.
- Round: offset($r=+3$) offset($\delta=-3$) rounds all outside (convex) corners, and leaves flat walls unchanged. However, walls less than $2*r$ thick vanish.

Parameters

r

Double. Amount to offset the polygon. When negative, the polygon is offset inwards. R specifies the radius of the circle that is rotated about the outline, either inside or outside. This mode produces rounded corners.

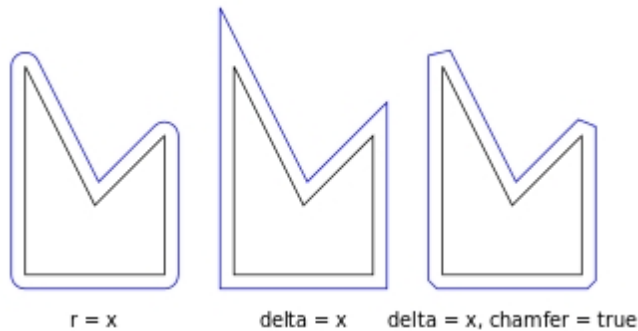
delta

Double. Amount to offset the polygon. When negative, the polygon is offset inwards. Delta specifies the distance of the new outline from the original outline, and therefore reproduces angled corners.

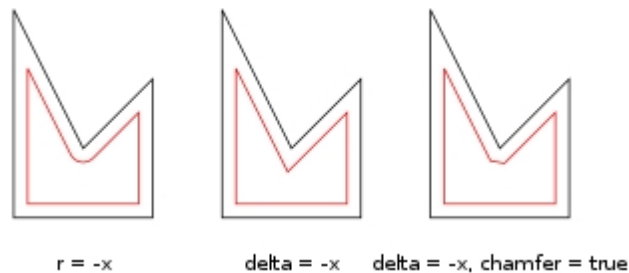
chamfer

Boolean. (default *false*) When using the delta parameter, this flag defines if edges should be chamfered (cut off with a straight line) or not (extended to their intersection).

Result for different parameters. The black polygon is the input for the offset() operation.



Positive r/delta value



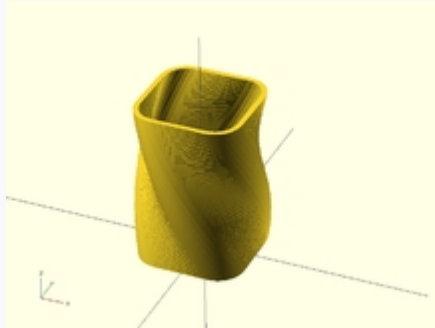
Negative r/delta value

Examples

```
// Example 1
linear_extrude(height = 60, twist = 90, slices = 60) {
  difference() {
    offset(r = 10) {
      square(20, center = true);
    }
    offset(r = 8) {
      square(20, center = true);
    }
  }
}
```

```
// Example 2
module fillet(r) {
  offset(r = -r) {
```

```
offset(delta = r) {
  children();
}
}
```



Example 1: Result.

Created with the Personal Edition of HelpNDoc: [Benefits of a Help Authoring Tool](#)

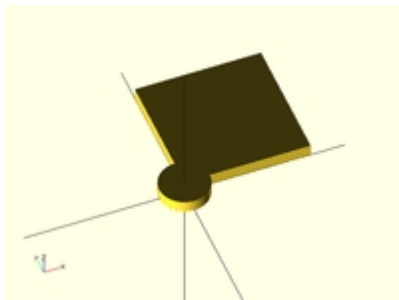
minkowski

Displays the **minkowski sum** of child nodes.

Usage example:

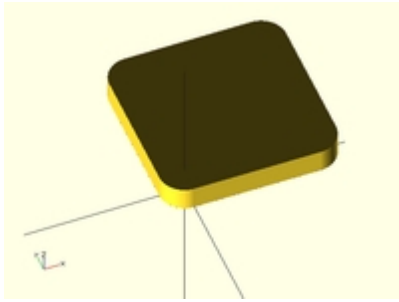
Say you have a flat box, and you want a rounded edge. There are multiple ways to do this (for example, see [hull](#) below), but minkowski is elegant. Take your box, and a cylinder:

```
$fn=50;
cube([10,10,1]);
cylinder(r=2,h=1);
```



Then, do a minkowski sum of them (note that the outer dimensions of the box are now $10+2+2 = 14$ units by 14 units by 2 units high as the heights of the objects are summed):

```
$fn=50;
minkowski()
{
  cube([10,10,1]);
  cylinder(r=2,h=1);
}
```



NB: The **origin** of the second object is used for the addition. If the second object is not centered, then the addition is asymmetric. The following minkowski sums are different: the first expands the original cube by 0.5 units in all directions, both positive and negative. The second expands it by +1 in each positive direction, but doesn't expand in the negative directions.

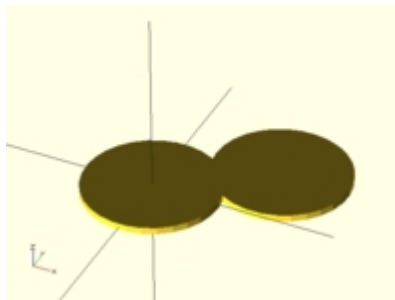
```
minkowski() {
  cube([10, 10, 1]);
  cylinder(1, center=true);
}
```

```
minkowski() {
  cube([10, 10, 1]);
  cylinder(1);
}
```

Warning: for high values of \$fn the minkowski sum may end up consuming lots of CPU and memory, since it has to combine every child node of each element with all the nodes of each other element. So if for example \$fn=100 and you combine two cylinders, then it does not just perform 200 operations as with two independent cylinders, but $100 \times 100 = 10000$ operations.

Created with the Personal Edition of HelpNDoc: [Write eBooks for the Kindle](#)

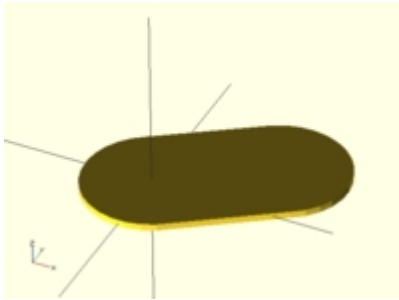
hull



Displays the **convex hull** of child nodes.

Usage example:

```
hull() {
  translate([15,10,0]) circle(10);
  circle(10);
}
```



The Hull of 2D objects uses their projections (shadows) on the xy plane, and produces a result on the xy plane. Their Z-height is not used in the operation.

Created with the Personal Edition of HelpNDoc: [Easily create Help documents](#)

Combining transformations

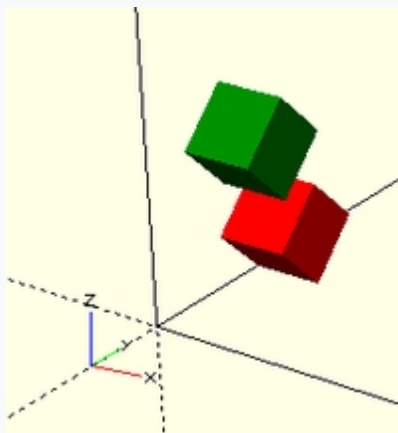
When combining transformations, it is a sequential process, but going right-to-left. That is

```
rotate( ... ) translate ( ... ) cube(5) ;
```

would first move the cube, and then move in an arc (turning it the same amount) at the radius given by the translation.

```
translate ( ... ) rotate( ... ) cube(5) ;
```

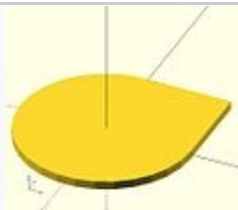
would first turn the cube and place it at the offset defined by the translate.



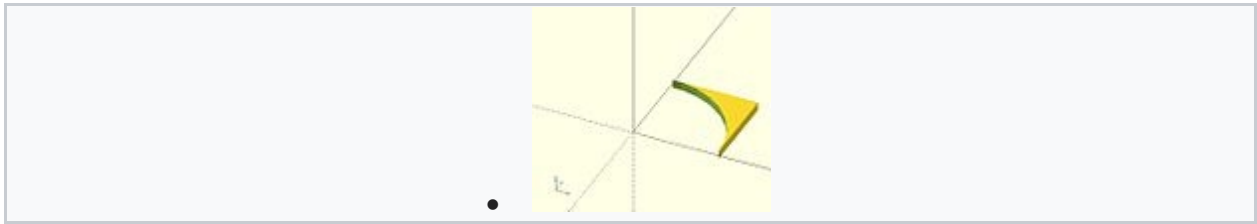
Created with the Personal Edition of HelpNDoc: [Write eBooks for the Kindle](#)

Boolean Combinations

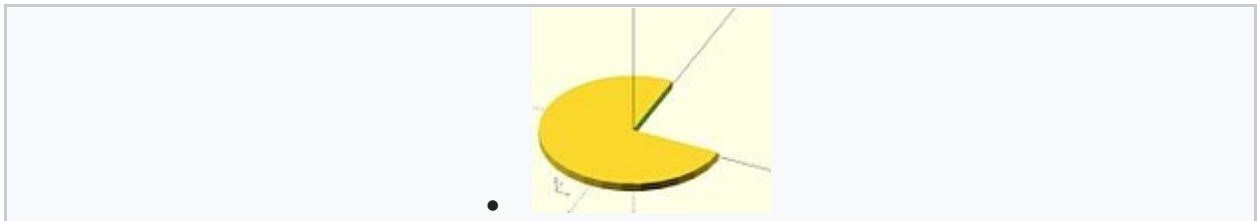
2D examples[\[edit\]](#)



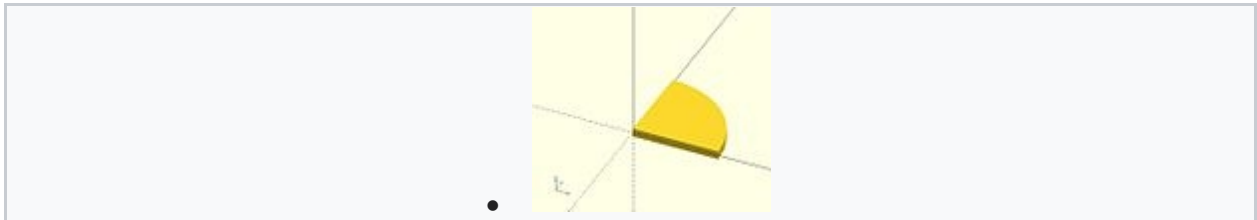
union (or)
circle + square



difference (and not)
square - circle



difference (and not)
circle - square



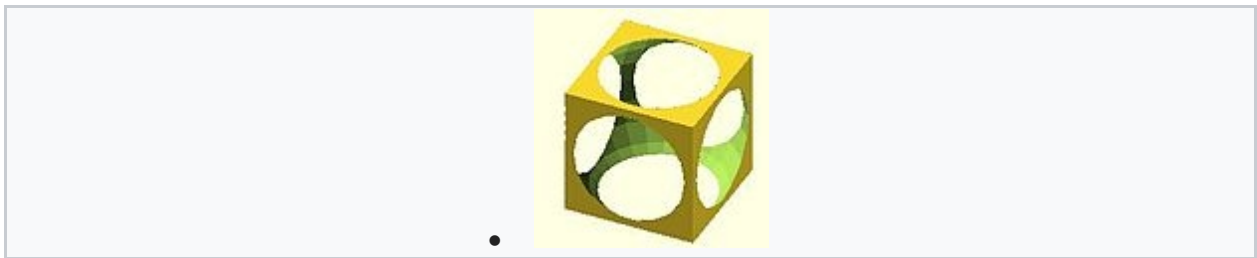
intersection (and)
circle - (circle - square)

```
union() {square(10);circle(10);} // square or circle
difference() {square(10);circle(10);} // square and not circle
difference() {circle(10);square(10);} // circle and not square
intersection(){square(10);circle(10);} // square and circle
```

3D examples[\[edit\]](#)



union (or)
sphere + cube



difference (and not)
cube - sphere



difference (and not)
sphere - cube

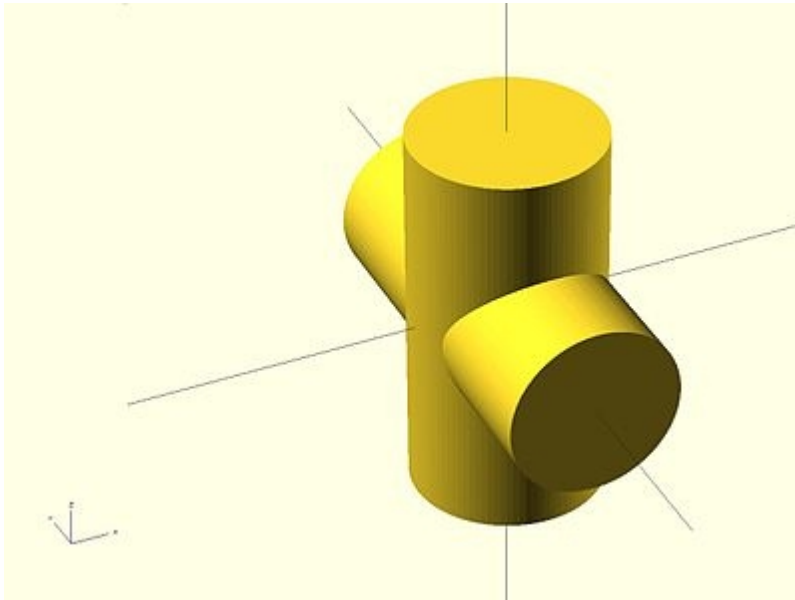


intersection (and)
sphere - (sphere - cube)

```
union() {cube(12, center=true); sphere(8);} // cube or sphere
difference() {cube(12, center=true); sphere(8);} // cube and not sphere
difference() {sphere(8); cube(12, center=true);} // sphere and not cube
intersection(){cube(12, center=true); sphere(8);} // cube and sphere
```

union

Creates a union of all its child nodes. This is the **sum** of all children (logical **or**). May be used with either 2D or 3D objects, but don't mix them.

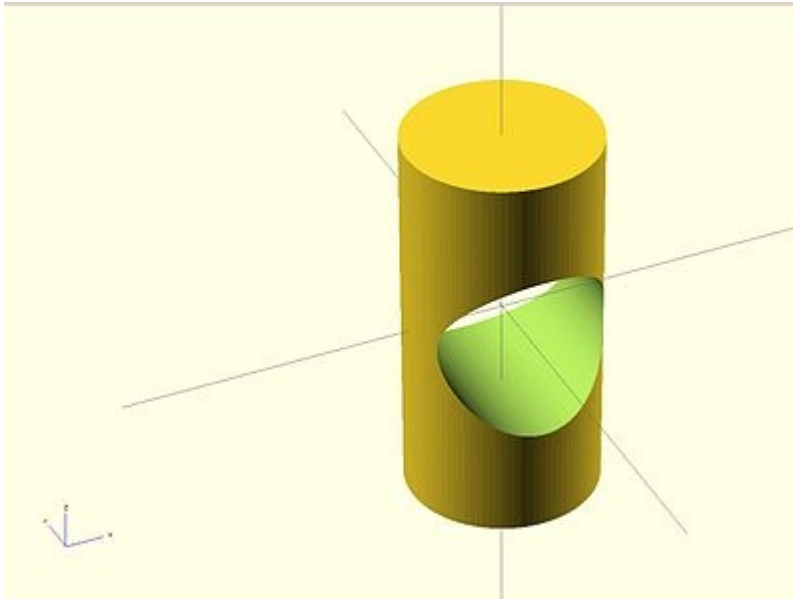


```
//Usage example:  
union() {  
  cylinder (h = 4, r=1, center = true, $fn=100);  
  rotate ([90,0,0]) cylinder (h = 4, r=0.9, center = true, $fn=100);  
}
```

Remark: union is implicit when not used. But it is mandatory, for example, in difference to group first child nodes into one.

difference

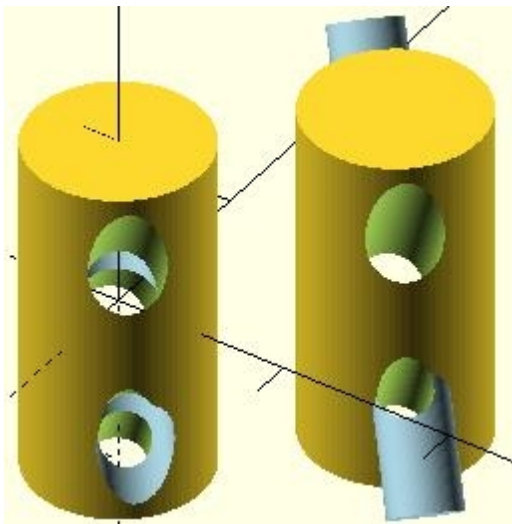
Subtracts the 2nd (and all further) child nodes from the first one (logical **and not**). May be used with either 2D or 3D objects, but don't mix them.



```
Usage example:
difference() {
  cylinder (h = 4, r=1, center = true, $fn=100);
  rotate ([90,0,0]) cylinder (h = 4, r=0.9, center = true, $fn=100);
}
```

difference with multiple children[\[edit\]](#)

Note, in the second instance, the result of adding a union of the 1st and 2nd children.



```
// Usage example for difference of multiple children:
$fn=90;
difference(){
  cylinder(r=5,h=20,center=true);
  rotate([00,140,-45]) color("LightBlue") cylinder(r=2,h=25,center=true);
  rotate([00,40,-50]) cylinder(r=2,h=30,center=true);
  translate([0,0,-10])rotate([00,40,-50]) cylinder(r=1.4,h=30,center=true);
}
// second instance with added union
translate([10,10,0]){
```



```

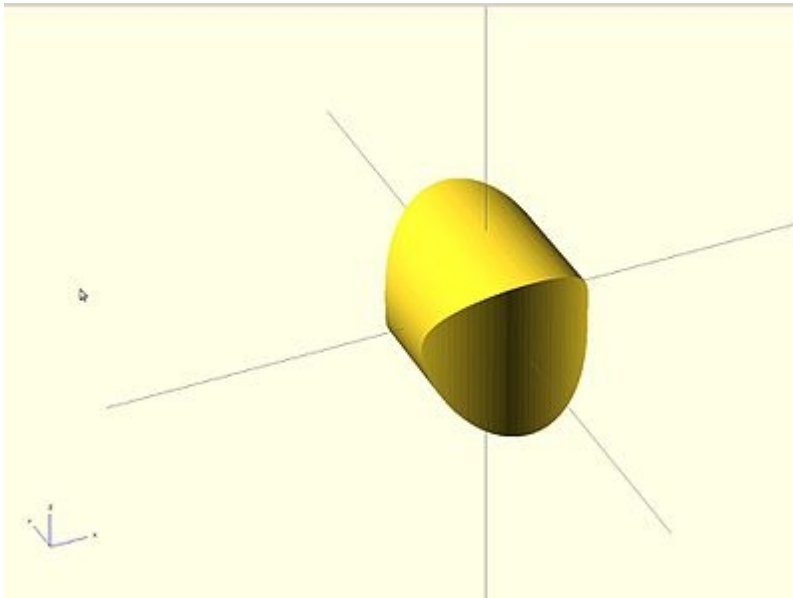
difference(){
union(){ // combine 1st and 2nd children
cylinder(r=5,h=20,center=true);
rotate([00,140,-45]) color("LightBlue") cylinder(r=2,h=25,center=true);
}
rotate([00,40,-50]) cylinder(r=2,h=30,center=true);
translate([0,0,-10])rotate([00,40,-50]) cylinder(r=1.4,h=30,center=true);
}
}

```

Created with the Personal Edition of HelpNDoc: [Free EPub and documentation generator](#)

intersection

Creates the intersection of all child nodes. This keeps the **overlapping** portion (logical **and**). Only the area which is common or shared by **all** children is retained. May be used with either 2D or 3D objects, but don't mix them.



```

//Usage example:
intersection() {
cylinder (h = 4, r=1, center = true, $fn=100);
rotate ([90,0,0]) cylinder (h = 4, r=0.9, center = true, $fn=100);
}

```

Created with the Personal Edition of HelpNDoc: [iPhone web sites made easy](#)

render

Warning: Using render, always calculates the CSG model for this tree (even in OpenCSG preview mode). This can make previewing very slow and OpenSCAD to appear to hang/freeze.

```

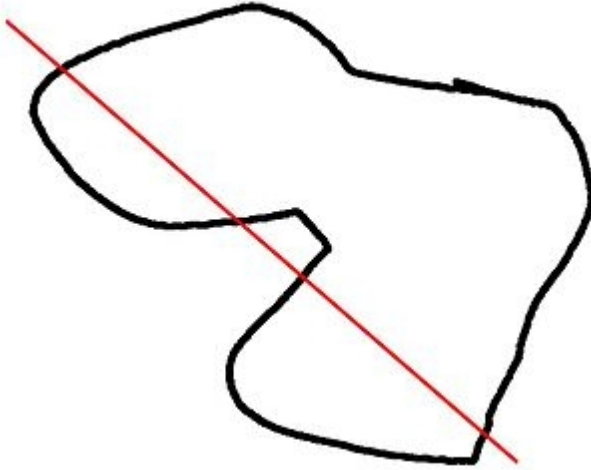
Usage example:
render(convexity = 1) { ... }

```

convexity

Integer. The convexity parameter specifies the maximum number of front and back sides a

ray intersecting the object might penetrate. This parameter is only needed for correctly displaying the object in OpenCSG preview mode and has no effect on the polyhedron rendering.



This image shows a 2D shape with a convexity of 4, as the ray indicated in red crosses the 2D shape a maximum of 4 times. The convexity of a 3D shape would be determined in a similar way. Setting it to 10 should work fine for most cases.

Created with the Personal Edition of HelpNDoc: [Write EPub books for the iPad](#)

Condition and Iterator functions

Created with the Personal Edition of HelpNDoc: [Free Kindle producer](#)

For loop

For loop [\[edit\]](#)

Evaluate each value in a range or vector, applying it to the following Action.

```
for(variable = [start : increment : end]) for(variable = [start : end]) for(variable = [vector])
```

parameters

As a range [start : <increment : > end] (see section on [range](#))

Note: For range, values are separated by colons rather than commas used in vectors.

start - initial value

increment or step - amount to increase the value, *optional*, *default* = 1

end - stop when next value would be past end

examples:

```
for (a =[3:5])echo(a); // 3 4 5 for (a =[3:0]){echo(a);} // 0 1 2
3 start > end is invalid, deprecated by 2015.3 for (a =[3:0.5:5])
echo(a); // 3 3.5 4 4.5 5 for (a =[0:2:5])echo(a); // 0 2 4 a
never equals end for (a =[3:-2:-1])echo(a); // 3 1 -1 negative
increment requires 2015.3 be sure end < start
```

As a vector

The Action is evaluated for each element of the vector

```
for (a =[3,4,1,5])echo(a); // 3 4 1 5 for (a =[0.3,PI,1,99])
{echo(a);} // 0.3 3.14159 1 99 x1=2; x2=8; x3=5.5; for (a
=[x1,x2,x3]){echo(a);} // 2 8 5.5 for (a =[1,2],6,"s",[3,4],
[5,6]])echo(a); // [1,2] 6 "s" [[3,4],[5,6]]
```

The vector can be described elsewhere, like 'for each' in other languages

```
animals = ["elephants", "snakes", "tigers", "giraffes"];
for(animal = animals) echo(str("I've been to the zoo and saw ",
animal)); // "I've been to the zoo and saw elephants", for each
animal
```

for() is an Operator. Operators require braces {} if more than one Action is within it scope. Actions end in semicolons, Operators do not.

for() is not an exception to the rule about variables having only one value within a scope. Each evaluation is given its own scope, allowing any variables to have unique values. No, you still can't do `a=a+1`;

Remember this is not an iterative language, the for() does not loop in the programmatic sense, it builds a tree of objects one branch for each item in the range/vector, inside each branch the 'variable' is a specific and separate instantiation or scope.

Hence:

```
for (i=[0:3]) translate([i*10,0,0]) cube(i+1);
```

Produces: *[See Design/Display-CSG-Tree menu]*

```
group() { group() { multmatrix([[1, 0, 0, 0], [0, 1, 0, 0], [0, 0,
1, 0], [0, 0, 0, 1]]) { cube(size = [1, 1, 1], center = false); }
multmatrix([[1, 0, 0, 10], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0,
1]]) { cube(size = [2, 2, 2], center = false); } multmatrix([[1,
0, 0, 20], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]]) { cube(size
= [3, 3, 3], center = false); } multmatrix([[1, 0, 0, 30], [0, 1,
0, 0], [0, 0, 1, 0], [0, 0, 0, 1]]) { cube(size = [4, 4, 4],
center = false); } } }
```

All instances of the for() exist at the same time, they do not iterate sequentially.

Nested for()

While it is reasonable to nest multiple for() statements such as:

```
for(z=[-180:45:+180]) for(x=[10:5:50]) rotate([0,0,z])
translate([x,0,0]) cube(1);
```

instead, all ranges/vectors can be included in the same for() operator.

```
for ( variable1 = <range or vector> , variable2 = <range or vector> ) <do something using both variables>
```

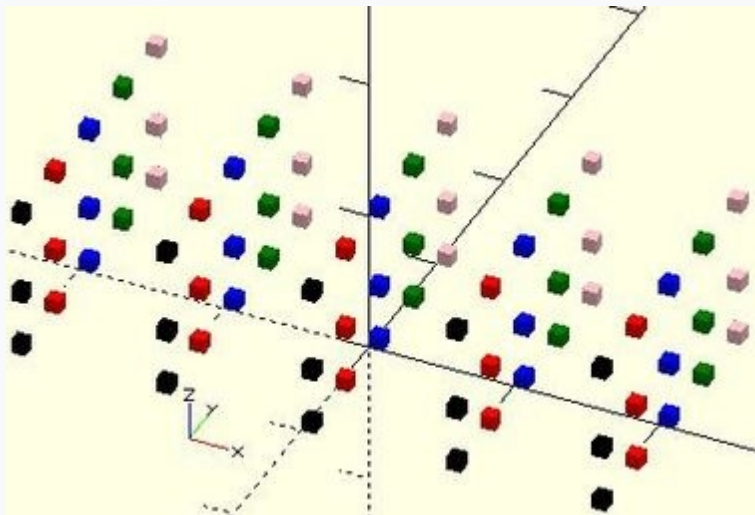
```
example for() nested 3 deep color_vec =
["black","red","blue","green","pink","purple"]; for (x = [-
20:10:20] ) for (y = [0:4] ) color(color_vec[y]) for (z =
[0,4,10] ) {translate([x,y*5-10,z])cube();} shorthand nesting for
same result color_vec =
["black","red","blue","green","pink","purple"]; for (x = [-
20:10:20], y = [0:4], z = [0,4,10] ) translate([x,y*5-10,z])
{color(color_vec[y])cube();}
```

Examples using vector of vectors

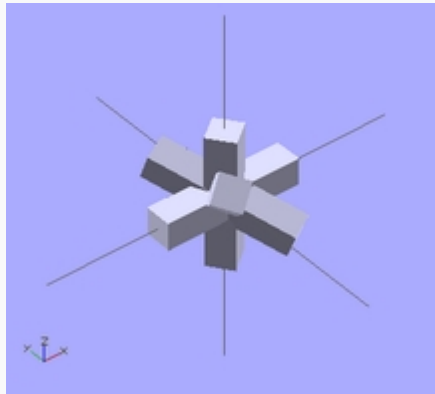
```
example 1 - iteration over a vector of vectors (rotation) for(i =
[ [ 0, 0, 0], [ 10, 20, 300], [200, 40, 57], [ 20, 88, 57] ])
{ rotate(i) cube([100, 20, 20], center = true); }
```

```
example 2 - iteration over a vector of vectors (translation) for(i
= [ [ 0, 0, 0], [10, 12, 10], [20, 24, 20], [30, 36, 30], [20, 48,
40], [10, 60, 50] ]) { translate(i) cube([50, 15, 10], center =
true); }
```

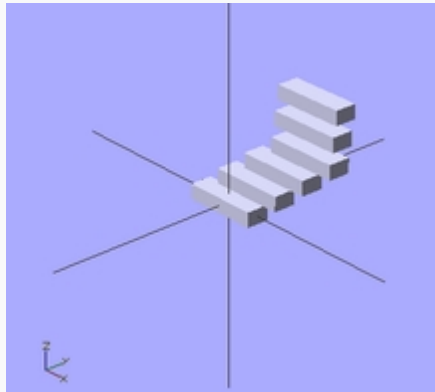
```
example 3 - iteration over a vector of vectors for(i = [ [[ 0, 0,
0], 20], [[10, 12, 10], 50], [[20, 24, 20], 70], [[30, 36, 30],
10], [[20, 48, 40], 30], [[10, 60, 50], 40] ]) { translate([i[0]
[0], 2*i[0][1], 0]) cube([10, 15, i[1]]); }
```



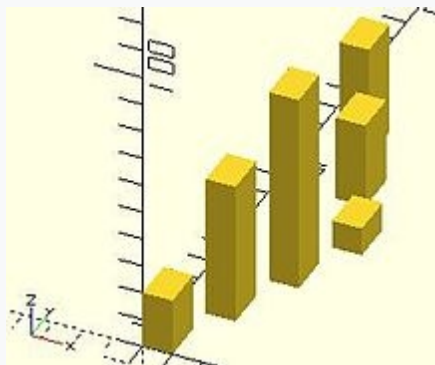
for() loops nested 3 deep



example 1 for() loop vector of vectors (rotation)



example 2 for() loop vector of vectors (translation)



example 3 for() loop vector of vectors

Created with the Personal Edition of HelpNDoc: [Easy to use tool to create HTML Help files and Help web sites](#)

Intersection For loop

Intersection For Loop [\[edit\]](#)

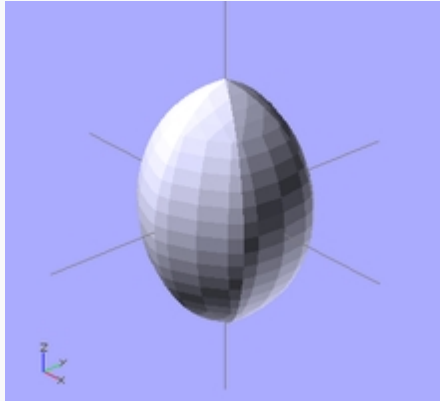
Iterate over the values in a range or vector and create the [intersection](#) of objects created by each pass.

Besides creating separate instances for each pass, the standard **for()** also groups all these instances creating an implicit union. **intersection_for()** is a work around because the implicit union prevents getting the expected results using a combination of the standard **for()** and **intersection()** statements.

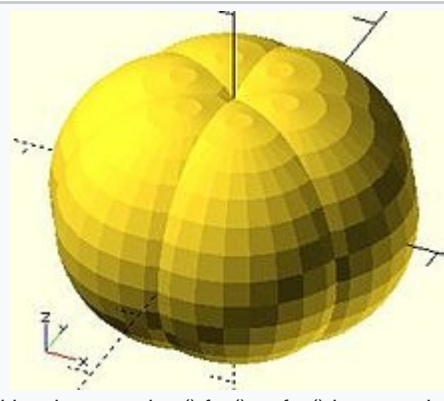
intersection_for() uses the same parameters, and works the same as a [For Loop](#), other than eliminating the implicit union.

example 1
- loop
over a
range:

```
intersection_for(
n =
[1 :
6])
{
rotate([0,
0, n
*
60])
{
translate(
[
5
,
0,0])
sphere(r=1
2);
}
}
```



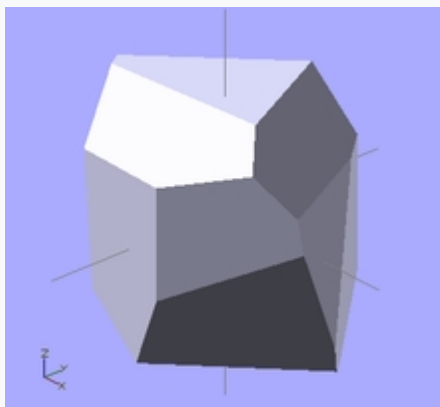
intersection_for()



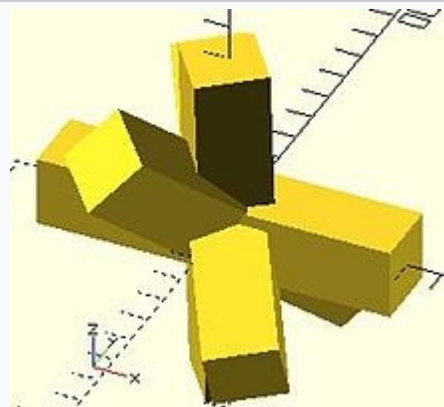
either intersection() for() or for() intersection()

example 2
- rotation :

```
intersection_for(
i =
[ [
0, 0,
0],
[ 10,
20,
300],
[200,
40,
57],
[ 20,
88,
57] ]
)
{
rotate(i)
cube(
[100,
20,
```



intersection_for()



intersection() for()

```
20],
cente
r =
true)
;
}
```

In

Created with the Personal Edition of HelpNDoc: [Free EPub and documentation generator](#)

if statement

Performs a test to determine if the actions in a sub scope should be performed or not.

```
if (test) scope1 if (test){scope1} if (test) scope1 else scope2 if
(test){scope1} else {scope2}
```

Parameters

test: Usually a boolean expression, but can be any value or variable.

[See here for true or false state of values.](#)

[See here for boolean and logical operators](#)

Do not confuse the assignment operator '=' with the equal operator '=='

scope1: one or more actions to take when test is **true**.

scope2: one or more actions to take when test is **false**.

```
if (b==a) cube(4); if (b<a) {cube(4); cylinder(6);} if (b&&a)
{cube(4); cylinder(6);} if (b!=a) cube(4); else cylinder(3); if
(b) {cube(4); cylinder(6);} else {cylinder(10,5,5);} if (!true)
{cube(4); cylinder(6);} else cylinder(10,5,5); if (x>y) cube(1,
center=false); else {cube(size = 2, center = true);} if (a==4) {}
else echo("a is not 4"); if ((b<5)&&(a>8)) {cube(4);} else
{cylinder(3);} if (b<5&&a>8) cube(4); else cylinder(3);
```

Since 2015.03 variables can now be assigned in any scope. Note that assignments are only valid within the scope in which they are defined - you are still not allowed to leak values to an outer scope. See [Scope of variables](#) for more details.

Nested if

The scopes of both the **if()** portion and the **else** portion, can in turn contain **if()** statements. This nesting can be to many depths.

```
if (test1) { scope1 if (test2) {scope2.1} else {scope2.2} } else
{ scope2 if (test3) {scope3.1} else {scope3.2} }
```

When scope1 and scope2 contain **only** the if() statement, the outer sets of braces can be removed.

```
if (test1) if (test2) {scope2.1} else {scope2.2} else if (test3)
{scope3.1} else {scope3.2}
```

Created with the Personal Edition of HelpNDoc: [Full-featured Kindle eBooks generator](#)

else if

```
if(test1) {scope1} else if(test2) {scope2} else if(test3) {scope3}
else if(test4) {scope4} else {scope5}
```

Note that *else* and *if* are two separate words. When working down the chain of tests, the first true uses its scope. All further tests are skipped.

example

```
if((k<8)&&(m>1)) cube(10); else if(y==6) {sphere(6);cube(10);}
else if(y==7) color("blue")sphere(5); else if(k+m!=8)
{cylinder(15,5,0);sphere(8);} else color("green")
{cylinder(12,5,0);sphere(8);}
```

Created with the Personal Edition of HelpNDoc: [Full-featured Help generator](#)

Conditional ?

A function that uses a test to determine which of 2 values to return.

```
a = test ? TrueValue : FalseValue ; echo( test ? TrueValue :
FalseValue );
```

Parameters

test: Usually a boolean expression, but can be any value or variable.

[See here for true or false state of values.](#)

[See here for boolean and logical operators](#)

Do not confuse assignment '=' with equal '=='

TrueValue: the value to return when test is **true**.

FalseValue: the value to return when test is **false**.

A value in OpenSCAD is either a Number (like 42), a Boolean (like true), a String (like "foo"), a Vector (like [1,2,3]), or the Undefined value (undef). Values can be stored in variables, passed as function arguments, and returned as function results.

This works like the ?: operator from the family of C-like programming languages.

Examples

```
a=1; b=2; c= a==b ? 4 : 5 ; // 5 a=1; b=2; c= a==b ? "a==b" :
"a!=b" ; // "a!=b" TrueValue = true; FalseValue = false; a=5; test
= a==1; echo( test ? TrueValue : FalseValue ); // false L = 75; R
= 2; test = (L/R)>25; TrueValue = [test,L,R,L/R,cos(30)];
FalseValue = [test,L,R,sin(15)]; a1 = test ? TrueValue :
FalseValue ; // [true, 75, 2, 37.5, 0.866025]
```

Created with the Personal Edition of HelpNDoc: [Easy EPub and documentation editor](#)

Recursive function calls

Recursive function calls are supported. Using the Conditional "... ? ... : ..." it's possible to ensure the recursion is terminated. Note: There is a built-in recursion limit to prevent an application crash. If the limit is hit, the function returns undef.

example


```
// recursion - find the sum of the values in a vector (array) by
// calling itself // from the start (or s'th element) to the i'th
// element - remember elements are zero based function sumv(v,i,s=0)
// = (i==s ? v[i] : v[i] + sumv(v,i-1,s)); vec=[ 10, 20, 30, 40 ];
// echo("sum vec=", sumv(vec,2,1)); // calculates 20+30=50
```

Some forms of [tail-recursion](#) elimination are supported.

Created with the Personal Edition of HelpNDoc: [Easily create iPhone documentation](#)

Let Statement

Set variables to a new value for a sub-tree. The parameters are evaluated sequentially and may depend on each other (as opposed to the deprecated assign() statement).

Parameters

The variables that should be set

example:

```
for (i = [10:50])
{
  let (angle = i*360/20, r= i*2, distance = r*5)
  {
    rotate(angle, [1, 0, 0])
    translate([0, distance, 0])
    sphere(r = r);
  }
}
```

Created with the Personal Edition of HelpNDoc: [Free EPub producer](#)

Mathematical Operators

Created with the Personal Edition of HelpNDoc: [Full-featured multi-format Help generator](#)

Scalar Arithmetical Operators

The scalar arithmetical operators take numbers as operands and produce a new number.

+	add
-	subtract
*	multiply
/	divide
%	modulo

The "-" can also be used as prefix operator to negate a number.

Created with the Personal Edition of HelpNDoc: [Easily create CHM Help documents](#)

Relational Operators

Relational operators produce a Boolean result from two operands.

<	less than
<=	less or equal
==	equal
!=	not equal
>=	greater or equal
>	greater than

If both operands are simple numbers, the meaning is self-evident.

If both operands are strings, alphabetical sorting determines equality and order. E.g., "ab" > "aa" > "a".

If both operands are Booleans, *true* > *false*. In an inequality comparison between a Boolean and a number *true* is treated as 1 and *false* is treated as 0. Other inequality tests involving Booleans return *false*.

If both operands are vectors, an equality test returns *true* when the vectors are identical and *false* otherwise. Inequality tests involving one or two vectors always return *false*, so for example `[1] < [2]` is *false*.

Dissimilar types always test as unequal with '==' and '!='. Inequality comparisons between dissimilar types, except for Boolean and numbers as noted above, always result in *false*. Note that `[1]` and `1` are different types so `[1] == 1` is *false*.

undef doesn't equal anything but *undef*. Inequality comparisons involving *undef* result in *false*.

nan doesn't equal anything (not even itself) and inequality tests all produce *false*. See [Numbers](#).

Created with the Personal Edition of HelpNDoc: [Create HTML Help, DOC, PDF and print manuals from 1 single source](#)

Logical Operators

All logical operators take Booleans as operands and produce a Boolean. Non-Boolean quantities are converted to Booleans before the operator is evaluated.

&&	logical AND
	logical OR
!	logical unary NOT

Since `[false]` is true, `false || [false]` is also true.

Note that how logical operators deal with vectors is different than relational operators:

`[1, 1] > [0, 2]` is *false*, but

`[false, false] && [false, false]` is true.

Created with the Personal Edition of HelpNDoc: [Produce electronic books easily](#)

Conditional Operator

The `?:` operator can be used to conditionally evaluate one or another expression. It works like

the `?:` operator from the family of C-like programming languages.

<code>?:</code>	Conditional operator
-----------------	----------------------

Usage Example:

```
a=1;
b=2;
c= a==b ? 4 : 5;
```

If `a` equals `b`, then `c` is set to 4, else `c` is set to 5.
The part "`a==b`" must be something that evaluates to a boolean value.

Created with the Personal Edition of HelpNDoc: [Single source CHM, PDF, DOC and HTML Help creation](#)

Vector-Number Operator

The vector-number operators take a vector and a number as operands and produce a new vector.

<code>*</code>	multiply all vector elements by number
<code>/</code>	divide all vector elements by number

Example

```
L = [1, [2, [3, "a"] ] ]; echo(5*L); // ECHO: [5, [10, [15,
undef]]]
```

Created with the Personal Edition of HelpNDoc: [Qt Help documentation made easy](#)

Vector Operators

The vector operators take vectors as operands and produce a new vector.

<code>+</code>	add element-wise
<code>-</code>	subtract element-wise

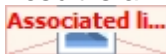
The `-` can also be used as prefix operator to element-wise negate a vector.

Example

```
L1 = [1, [2, [3, "a"] ] ]; L2 = [1, [2, 3] ]; echo(L1+L1); //
ECHO: [2, [4, [6, undef]]] echo(L1+L2); // ECHO: [2, [4, undef]]
```

Created with the Personal Edition of HelpNDoc: [iPhone web sites made easy](#)

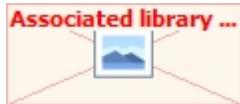
Vector Dot-Product Operator

If both operands of multiplication are simple vectors, the result is a number according to the linear algebra rule for **dot product**. `c = u*v;` results in . If the operands' sizes don't match, the result is `undef`.

Matrix Multiplication

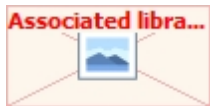
If one or both operands of multiplication are matrices, the result is a simple vector or matrix according to the linear algebra rules for **matrix product**. In the following, A, B, C... are matrices, u, v, w... are vectors. Subscripts i, j denote element indices.

For A a matrix of size $n \times m$ and B a matrix of size $m \times p$, their product $C = A * B$ is a matrix of size $n \times p$ with elements



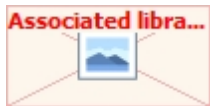
$C = B * A$; results in `undef` unless $n = p$.

For A a matrix of size $n \times m$ and v a vector of size m, their product $u = A * v$ is a vector of size n with elements



In linear algebra, this is the **product of a matrix and a column vector**.

For v a vector of size n and A a matrix of size $n \times m$, their product $u = v * A$ is a vector of size m with elements



In linear algebra, this is the product of a row vector and a matrix.

Matrix multiplication is not commutative: , .

Trigonometric Functions

cos

The trig functions use the C Language mathematics functions, which are based in turn on Binary Floating Point mathematics, which use approximations of Real Numbers during calculation. OpenSCAD's math functions use the C++ 'double' type, inside Value.h/Value.cc,

A good resource for the specifics of the C library math functions, such as valid inputs/output ranges, can be found at the Open Group website [math.h](#) & [acos](#)

cos[\[edit\]](#)

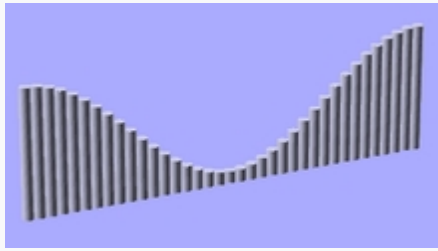
Mathematical **cosine** function of degrees. See [Cosine](#)

Parameters**<degrees>**

Decimal. Angle in degrees.

Usage Example:

```
for(i=[0:36])
  translate([i*
    10,0,0])
  cylinder(r=5,
    h=cos(i*10)
    *50+60);
```



Created with the Personal Edition of HelpNDoc: [Produce online help for Qt applications](#)

sin

Mathematical **sine** function. See [Sine](#)

Parameters**<degrees>**

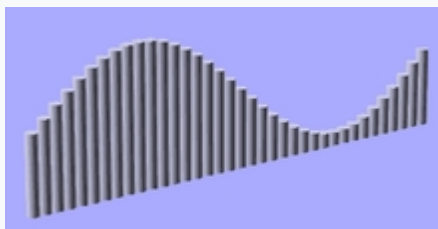
Decimal. Angle in degrees.

Usage example 1:

```
for (i = [0:5]) {
  echo(360*i/6, sin(360*i/6)
    *80, cos(360*i/6)*80);
  translate([sin(360*i/6)*80,
    cos(360*i/6)*80, 0 ])
  cylinder(h = 200, r=10);
}
```

Usage example 2:

```
for(i=[0:36])
  translate([i*
    10,0,0])
  cylinder(r=5,
    h=sin(i*10)
    *50+60);
```



OpenSCAD Sin Function

Created with the Personal Edition of HelpNDoc: [Full-featured Documentation generator](#)

tan

Mathematical **tangent** function. See [Tangent](#)

Parameters**<degrees>**

Decimal. Angle in degrees.

Usage example:

```
for (i = [0:5]) {
  echo(360*i/6, tan(360*i/6)*80);
  translate([tan(360*i/6)*80, 0, 0 ])
  cylinder(h = 200, r=10);
}
```

Created with the Personal Edition of HelpNDoc: [Easily create EPub books](#)

acos

Mathematical **arccosine**, or **inverse cosine**, expressed in degrees. See: [Inverse trigonometric functions](#)

Created with the Personal Edition of HelpNDoc: [Free EBook and documentation generator](#)

asin

Mathematical **arcsine**, or **inverse sine**, expressed in degrees. See: [Inverse trigonometric functions](#)

Created with the Personal Edition of HelpNDoc: [iPhone web sites made easy](#)

atan

Mathematical **arctangent**, or **inverse tangent**, function. Returns the principal value of the arc tangent of x, expressed in degrees. See: [Inverse trigonometric functions](#)

Created with the Personal Edition of HelpNDoc: [Free Qt Help documentation generator](#)

atan2

Mathematical **two-argument atan** function `atan2(y,x)` that spans the full 360 degrees. This function returns the full angle (0-360) made between the x axis and the vector(x,y) expressed in degrees. `atan` can not distinguish between y/x and $-y/-x$ and returns angles from -90 to +90
See: [atan2](#)

Usage examples:

```
atan2(5.0,-5.0); //result: 135 degrees. atan() would give -45
atan2(y,x); //angle between (1,0) and (x,y) = angle around z-axis
```

Created with the Personal Edition of HelpNDoc: [Produce online help for Qt applications](#)

Other Mathematical Functions

Created with the Personal Edition of HelpNDoc: [Generate Kindle eBooks with ease](#)

abs

Mathematical absolute value function. Returns the positive value of a signed decimal number.

Usage examples:

`abs(-5.0);` returns 5.0

```
abs(0);    returns 0.0
abs(8.0);  returns 8.0
```

Created with the Personal Edition of HelpNDoc: [Full-featured multi-format Help generator](#)

ceil

Mathematical ceiling function.

Returns the next highest integer value by rounding up value if necessary.

See: Ceil Function

```
echo(ceil(4.4),ceil(-4.4));    // produces ECHO: 5, -4
```

Created with the Personal Edition of HelpNDoc: [Free CHM Help documentation generator](#)

concat

Return a new vector that is the result of appending the elements of the supplied vectors.

Where an argument is a vector the elements of the vector are individually appended to the result vector. Strings are distinct from vectors in this case.

Usage examples:

```
echo(concat("a","b","c","d","e","f"));    // produces ECHO: ["a", "b", "c", "d", "e", "f"]
echo(concat(["a","b","c"],["d","e","f"]));  // produces ECHO: ["a", "b", "c", "d", "e", "f"]
echo(concat(1,2,3,4,5,6));                // produces ECHO: [1, 2, 3, 4, 5, 6]
Vector of vectors
```

```
echo(concat([ [1],[2] ], [ [3] ] ));        // produces ECHO: [[1], [2], [3]]
```

Contrast with strings

```
echo(concat([1,2,3],[4,5,6]));              // produces ECHO: [1, 2, 3, 4, 5, 6]
echo(concat("abc","def"));                 // produces ECHO: ["abc", "def"]
echo(str("abc","def"));                    // produces ECHO: "abcdef"
```

Created with the Personal Edition of HelpNDoc: [Free help authoring environment](#)

cross

Calculates the cross product of two vectors in 3D space. The result is a vector that is perpendicular to both of the input vectors.

Using invalid input parameters (e.g. vectors with a length different from 3 or other types) produces an undefined result.

Usage examples:

```
echo(cross([2, 3, 4], [5, 6, 7]));          // produces ECHO: [-3, 6, -3]
echo(cross([2, 1, -3], [0, 4, 5]));         // produces ECHO: [17, -10, 8]
echo(cross([2, 3, 4], "5"));               // produces ECHO: undef
```

Created with the Personal Edition of HelpNDoc: [Free iPhone documentation generator](#)

exp

Mathematical exp function. Returns the base-e exponential function of x, which is the number e raised to the power x. See: Exponent

```
echo(exp(1),exp(ln(3)*4)); // produces ECHO: 2.71828, 81
```

Created with the Personal Edition of HelpNDoc: [Free Web Help generator](#)

floor

Mathematical **floor** function. $\text{floor}(x)$ = is the largest integer not greater than x

See: [Floor Function](#)

```
echo(floor(4.4),floor(-4.4)); // produces ECHO: 4, -5
```

Created with the Personal Edition of HelpNDoc: [Create cross-platform Qt Help files](#)

ln

Mathematical **natural logarithm**. See: [Natural logarithm](#)

Created with the Personal Edition of HelpNDoc: [Benefits of a Help Authoring Tool](#)

len

Mathematical length function. Returns the length of an array, a vector or a string parameter.

Usage examples:

```
str1="abcdef"; len_str1=len(str1);
echo(str1,len_str1);
```

```
a=6; len_a=len(a);
echo(a,len_a);
```

```
array1=[1,2,3,4,5,6,7,8]; len_array1=len(array1);
echo(array1,len_array1);
```

```
array2=[[0,0],[0,1],[1,0],[1,1]]; len_array2=len(array2);
echo(array2,len_array2);
```

```
len_array2_2=len(array2[2]);
echo(array2[2],len_array2_2);
```

Results:

```
ECHO: "abcdef", 6
ECHO: 6, undef
ECHO: [1, 2, 3, 4, 5, 6, 7, 8], 8
ECHO: [[0, 0], [0, 1], [1, 0], [1, 1]], 4
ECHO: [1, 0], 2
```

This function allows (e.g.) the parsing of an array, a vector or a string.

Usage examples:

```
str2="4711";
for (i=[0:len(str2)-1])
    echo(str("digit ",i+1," : ",str2[i]));
```

Results:

```
ECHO: "digit 1 : 4"
```



```
ECHO: "digit 2 : 7"
ECHO: "digit 3 : 1"
ECHO: "digit 4 : 1"
```

Note that the `len()` function is not defined when a simple variable is passed as the parameter.

This is useful when handling parameters to a module, similar to how shapes can be defined as a single number, or as an `[x,y,z]` vector; i.e. `cube(5)` or `cube([5,5,5])`

For example

```
module dolt(size) {
    if (len(size) == undef) {
        // size is a number, use it for x,y & z. (or could be undef)
        do([size,size,size]);
    } else {
        // size is a vector, (could be a string but that would be stupid)
        do(size);
    }
}

dolt(5); // equivalent to [5,5,5]
dolt([5,5,5]); // similar to cube(5) vs cube([5,5,5])
```

Created with the Personal Edition of HelpNDoc: [Easily create CHM Help documents](#)

let

Sequential assignment of variables inside an expression. The following expression is evaluated in context of the `let` assignments and can use the variables. This is mainly useful to make complicated expressions more readable by assigning interim results to variables.

Parameters

`let (var1 = value1, var2 = f(var1), var3 = g(var1, var2)) expression`

Usage Example:

```
echo(let(a = 135, s = sin(a), c = cos(a)) [ s, c ]); // ECHO: [0.707107, -0.707107]
```

Created with the Personal Edition of HelpNDoc: [What is a Help Authoring tool?](#)

log

Mathematical logarithm to the base 10. Example: $\log(1000) = 3$. See: [Logarithm](#)

Created with the Personal Edition of HelpNDoc: [Easy EPub and documentation editor](#)

lookup

Look up value in table, and linearly interpolate if there's no exact match. The first argument is the value to look up. The second is the lookup table -- a vector of key-value pairs.

Parameters

key

A lookup key

<key,value> array

keys and values

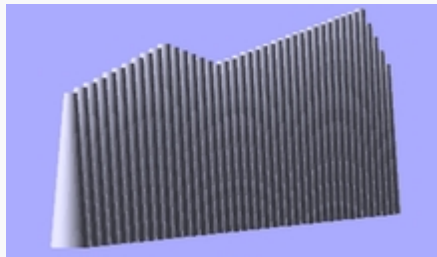
Notes

There is a bug in which out-of-range keys return the first value in the list. Newer versions of Openscad should use the top or bottom end of the table as appropriate instead.

Usage example:

- Create a 3D chart made from cylinders of different heights.

```
function get_cylinder_h(p) = lookup(p, [
[ -200, 5 ],
[ -50, 20 ],
[ -20, 18 ],
[ +80, 25 ],
[ +150, 2 ]
]);
for (i = [-100:5:+100]) {
// echo(i, get_cylinder_h(i));
translate([ i, 0, -30 ]) cylinder(r1 = 6, r2 = 2, h = get_cylinder_h(i)
*3);
}
```



OpenSCAD Lookup Function

Created with the Personal Edition of HelpNDoc: [Full-featured multi-format Help generator](#)

max

Returns the maximum of the parameters. If a single vector is given as parameter, returns the maximum element of that vector.

Parameters

max(n,n{n},n)...

max(vector)

<n>

Two or more decimals

<vector>

Single vector of decimals [Note: Requires version 2014.06].

Usage Example:

max(3.0,5.0)

max(8.0,3.0,4.0,5.0)

max([8,3,4,5])

Results:

5

8

8

Created with the Personal Edition of HelpNDoc: [Benefits of a Help Authoring Tool](#)

min

Returns the minimum of the parameters. If a single vector is given as parameter, returns the minimum element of that vector.

Parameters

`min(n,n{n},n,...)`

`min(vector)`

`<n>`

Two or more decimals

`<vector>`

Single vector of decimals [Note: Requires version 2014.06].

Usage Example:

```
min(3.0,5.0)
```

```
min(8.0,3.0,4.0,5.0)
```

```
min([8,3,4,5])
```

Results:

3

3

3

Created with the Personal Edition of HelpNDoc: [Full-featured multi-format Help generator](#)

norm

Returns the euclidean norm of a vector. Note this returns the actual numeric length while `len` returns the number of elements in the vector or array.

Usage examples:

```
a=[1,2,3,4];
```

```
b="abcd";
```

```
c=[];
```

```
d="";
```

```
e=[[1,2,3,4],[1,2,3],[1,2],[1]];
```

```
echo(norm(a)); //5.47723
```

```
echo(norm(b)); //undef
```

```
echo(norm(c)); //0
```

```
echo(norm(d)); //undef
```

```
echo(norm(e[0])); //5.47723
```

```
echo(norm(e[1])); //3.74166
```

```
echo(norm(e[2])); //2.23607
```

```
echo(norm(e[3])); //1
```

Results:

ECHO: 5.47723

ECHO: undef

ECHO: 0

ECHO: undef

ECHO: 5.47723

ECHO: 3.74166

ECHO: 2.23607

ECHO: 1

Created with the Personal Edition of HelpNDoc: [Write EPub books for the iPad](#)

pow

Mathematical power function.

Parameters

<base>

Decimal. Base.

<exponent>

Decimal. Exponent.

Usage examples:

```
for (i = [0:5]) {
  translate([i*25,0,0]) {
    cylinder(h = pow(2,i)*5, r=10);
    echo (i, pow(2,i));
  }
}
echo(pow(10,2)); // means 10^2 or 10*10
// result: ECHO: 100
```

```
echo(pow(10,3)); // means 10^3 or 10*10*10
// result: ECHO: 1000
```

```
echo(pow(125,1/3)); // means 125^(0.333...), which calculates the cube root of 125
// result: ECHO: 5
```

Created with the Personal Edition of HelpNDoc: [Free EPub producer](#)

rand

Random number generator. Generates a constant vector of pseudo random numbers, much like an array. The numbers are doubles not integers. When generating only one number, you still call it with `variable[0]`

Parameters

`min_value`

Minimum value of random number range

`max_value`

Maximum value of random number range

`value_count`

Number of random numbers to return as a vector

`seed_value` (optional)Seed value for random number generator for repeatable results. On versions before late 2015, `seed_value` gets rounded to the nearest integer

Usage Examples:

```
// get a single number
single_rand = rand(0,10,1)[0];
echo(single_rand);
// get a vector of 4 numbers
seed=42;
random_vect=rand(5,15,4,seed);
echo("Random Vector: ",random_vect);
sphere(r=5);
for(i=[0:3]) {
  rotate(360*i/4) {
    translate([10+random_vect[i],0,0])
    sphere(r=random_vect[i]/2);
  }
}
// ECHO: "Random Vector: ", [8.7454, 12.9654, 14.5071, 6.83435]
```

Created with the Personal Edition of HelpNDoc: [Free help authoring environment](#)

round

The "round" operator returns the greatest or least integer part, respectively, if the numeric input is positive or negative.

Some examples:

$\text{round}(x.5) = x+1.$

$\text{round}(x.49) = x.$

$\text{round}(-(x.5)) = -(x+1).$

$\text{round}(-(x.49)) = -x.$

`round(5.4); //-> 5`

`round(5.5); //-> 6`

`round(5.6); //-> 6`

Created with the Personal Edition of HelpNDoc: [What is a Help Authoring tool?](#)

sign

Mathematical signum function. Returns a unit value that extracts the sign of a value see: Signum function

Parameters

<x>

Decimal. Value to find the sign of.

Usage examples:

`sign(-5.0);`

`sign(0);`

`sign(8.0);`

Results:

-1.0

0.0

1.0

Created with the Personal Edition of HelpNDoc: [What is a Help Authoring tool?](#)

sqrt

Mathematical square root function.

Usage Examples:

`translate([sqrt(100),0,0])sphere(100);`

Created with the Personal Edition of HelpNDoc: [Easy to use tool to create HTML Help files and Help web sites](#)

Infinites and NaNs

How does OpenSCAD deal with inputs like (1/0)? Basically, the behavior is inherited from the language

OpenSCAD was written in, the C++ language, and its floating point number types and the associated C math library. This system allows representation of both positive and negative infinity by the special values "Inf" or "-Inf". It also allow representation of creatures like $\sqrt{-1}$ or $0/0$ as "NaN", an abbreviation for "Not A Number". Some very nice explanations can be found on the web, for example the Open Group's site on math.h or Wikipedia's page on the IEEE 754 number format. However OpenSCAD is it's own language so it may not exactly match everything that happens in C. For example, OpenSCAD uses degrees instead of radians for trigonometric functions. Another example is that $\sin()$ does not throw a "domain error" when the input is $1/0$, although it does return NaN.

Here are some examples of infinite input to OpenSCAD math functions and the resulting output, taken from OpenSCAD's regression test system in late 2015.

0/0: nan	$\sin(1/0)$: nan	$\text{asin}(1/0)$: nan	$\ln(1/0)$: inf	$\text{round}(1/0)$: inf
-0/0: nan	$\cos(1/0)$: nan	$\text{acos}(1/0)$: nan	$\ln(-1/0)$: nan	$\text{round}(-1/0)$: -inf
0/-0: nan	$\tan(1/0)$: nan	$\text{atan}(1/0)$: 90	$\log(1/0)$: inf	$\text{sign}(1/0)$: 1
1/0: inf	$\text{ceil}(-1/0)$: -inf	$\text{atan}(-1/0)$: -90	$\log(-1/0)$: nan	$\text{sign}(-1/0)$: -1
1/-0: -inf	$\text{ceil}(1/0)$: inf	$\text{atan2}(1/0, -1/0)$: 135	$\max(-1/0, 1/0)$: inf	$\sqrt{1/0}$: inf
-1/0: -inf	$\text{floor}(-1/0)$: -inf	$\exp(1/0)$: inf	$\min(-1/0, 1/0)$: -inf	$\sqrt{-1/0}$: nan
-1/-0: inf	$\text{floor}(1/0)$: inf	$\exp(-1/0)$: 0	$\text{pow}(2, 1/0)$: inf	$\text{pow}(2, -1/0)$: 0

Created with the Personal Edition of HelpNDoc: [Produce online help for Qt applications](#)

String Functions

Created with the Personal Edition of HelpNDoc: [What is a Help Authoring tool?](#)

str

Convert all arguments to strings and concatenate.

Usage examples:

```
number=2;
echo ("This is ",number,3," and that's it.");
echo (str("This is ",number,3," and that's it."));
Results:
```

ECHO: "This is ", 2, 3, " and that's it."

ECHO: "This is 23 and that's it."

Created with the Personal Edition of HelpNDoc: [Write EPub books for the iPad](#)

chr

Convert numbers to a string containing character with the corresponding code. OpenSCAD uses Unicode, so the number is interpreted as Unicode code point. Numbers outside the valid code point range produce an empty string.

Parameters

`chr(Number)`

Convert one code point to a string of length 1 (number of bytes depending on UTF-8 encoding) if the code point is valid.

`chr(Vector)`

Convert all code points given in the argument vector to a string.

`chr(Range)`

Convert all code points produced by the range argument to a string.

Examples

```
echo(chr(65), chr(97));    // ECHO: "A", "a"
echo(chr(65, 97));        // ECHO: "Aa"
echo(chr([66, 98]));      // ECHO: "Bb"
echo(chr([97 : 2 : 102])); // ECHO: "ace"
echo(chr(-3));            // ECHO: ""
echo(chr(9786), chr(9788)); // ECHO: " ", " "
echo(len(chr(9788)));      // ECHO: 1
```

Note: When used with `echo()` the output to the console for character codes greater than 127 is platform dependent.

Created with the Personal Edition of HelpNDoc: [Create cross-platform Qt Help files](#)

ord

Convert a character to a number representing the Unicode code point. If the parameter is not a string, the `ord()` returns `undef`.

Parameters

`ord(String)`

Convert the first character of the given string to a Unicode code point.

Examples

```
echo(ord("a"));
// ECHO: 97
```

```
echo(ord("BCD"));
// ECHO: 66
```

```
echo([for (c = "Hello! ") ord(c)]);
// ECHO: [72, 101, 108, 108, 111, 33, 32, 128578]
```

Created with the Personal Edition of HelpNDoc: [Free PDF documentation generator](#)

Also see `search()`

Created with the Personal Edition of HelpNDoc: [Write eBooks for the Kindle](#)

List Comprehensions

Created with the Personal Edition of HelpNDoc: [Single source CHM, PDF, DOC and HTML Help creation](#)

Basic Syntax

The list comprehensions provide a flexible way to generate lists using the general syntax

[list-definition expression]

The following elements are supported to construct the list definition

for (i = sequence)
 Iteration over a range or an existing list
 for (init;condition;next)
 Simple recursive call represented as C-style for
 each
 Takes a sequence value as argument, and adds each element to the list being constructed. each x is equivalent to `for (i = x) i`
 if (condition)
 Selection criteria, when true the expression is calculated and added to the result list
 let (x = value)
 Local variable assignment

Created with the Personal Edition of HelpNDoc: [Free HTML Help documentation generator](#)

Multiple generator expressions

The list comprehension syntax is generalized to allow multiple expressions. This allows to easily construct lists from multiple sub lists generated by different list comprehension expressions avoiding concat.

```
steps = 50;
```

```
points = [
    // first expression generating the points in the positive Y quadrant
    for (a = [0 : steps]) [ a, 10 * sin(a * 360 / steps) + 10 ],
    // second expression generating the points in the negative Y quadrant
    for (a = [steps : -1 : 0]) [ a, 10 * cos(a * 360 / steps) - 20 ],
    // additional list of fixed points
    [ 10, -3 ], [ 3, 0 ], [ 10, 3 ]
];
```

```
polygon(points);
```

Created with the Personal Edition of HelpNDoc: [Free Web Help generator](#)

for

The for element defines the input values for the list generation. The syntax is the same as used by the [for](#) iterator. The sequence to the right of the equals sign can be any list. The for element iterates over all the members of the list. The variable on the left of the equals sign take on the value of each member of the sequence in turn. This value can then be processed in the child of the for element, and each result becomes a member of the final list that is produced.

If the sequence has more than one dimension, for iterates over the first dimension only. Deeper dimensions can be accessed by nesting for elements.

Several common usage patterns are presented here.

[for (i = [start : step : end]) i]

Generate output based on a range definition, this version is mainly useful to calculate list values or access existing lists using the range value as index.

Examples

```
// generate a list with all values defined by a range
list1 = [ for (i = [0 : 2 : 10]) i ];
echo(list1); // ECHO: [0, 2, 4, 6, 8, 10]
```

```
// extract every second character of a string
str = "SomeText";
```



```
list2 = [ for (i = [0 : 2 : len(str) - 1]) str[i] ];
echo(list2); // ECHO: ["S", "m", "T", "x"]
```

```
// indexed list access, using function to map input values to output
values
function func(x) = x < 1 ? 0 : x + func(x - 1);
input = [1, 3, 5, 8];
output = [for (a = [ 0 : len(input) - 1 ]) func(input[a]) ];
echo(output); // ECHO: [1, 6, 15, 36]
```

[for (i = [a, b, c, ...]) i]

Use list parameter as input, this version can be used to map input values to calculated output values.

Examples

```
// iterate over an existing list
friends = ["John", "Mary", "Alice", "Bob"];
list = [ for (i = friends) len(i) ];
echo(list); // ECHO: [4, 4, 5, 3]
```

```
// map input list to output list
list = [ for (i = [2, 3, 5, 7, 11]) i * i ];
echo(list); // ECHO: [4, 9, 25, 49, 121]
```

```
// calculate Fibonacci numbers
function func(x) = x < 3 ? 1 : func(x - 1) + func(x - 2);
input = [7, 10, 12];
output = [for (a = input) func(a) ];
echo(output); // ECHO: [13, 55, 144]
```

[for (c = "String") c]

Generate output based on a string, this iterates over each character of the string.

[Note: Requires version 2019.05]

Examples

```
echo([ for (c = "String") c ]);
// ECHO: ["S", "t", "r", "i", "n", "g"]
```

[for (a = inita, b = initb, ...;condition;a = nexta, b = nextb, ...) expr]

Generator for expressing simple recursive call in a c-style for loop.

[Note: Requires version 2019.05]

The recursive equivalent of this generator is

```
function f(a, b, ...) = condition ? concat([expr], f(nexta, nextb, ...)) :
[]; f(inita, initb, ...)
```

Examples

```
echo( [for (a = 0, b = 1;a < 5;a = a + 1, b = b + 2) [ a, b * b ] ] );
// ECHO: [[0, 1], [1, 9], [2, 25], [3, 49], [4, 81]]
```

```
// Generate fibonacci sequence
echo([for (a = 0, b = 1; a < 1000; x = a + b, a = b, b = x) a]);
// ECHO: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987]
```

Created with the Personal Edition of HelpNDoc: [Easily create EPub books](#)

each

each embeds the values of a list given as argument directly, effectively unwrapping the argument list.

```
// Without using "each", a nested list is generated
echo([ for (a = [1 : 4]) [a, a * a] ]);
// ECHO: [[1, 1], [2, 4], [3, 9], [4, 16]]
// Adding "each" unwraps the inner list, producing a flat list as result
echo([ for (a = [1 : 4]) each [a, a * a] ]);
// ECHO: [1, 1, 2, 4, 3, 9, 4, 16]
```

each unwraps ranges and helps to build more general *for* lists when combined with multiple generator expressions.

```
A = [-2, each [1:2:5], each [6:-2:0], -1];
echo([ for (a = A) 2 * a ]);
// ECHO: [-4, 2, 6, 10, 12, 8, 4, 0, -2]
```

Created with the Personal Edition of HelpNDoc: [Create help files for the Qt Help Framework](#)

if

The *if* element allows selection if the expression should be allocated and added to the result list or not. In the simplest case this allows filtering of an list.

[for (i = list) if (condition(i)) i]

When the evaluation of the condition returns true, the expression *i* is added to the result list.

Example

```
list = [ for (a = [ 1 : 8 ]) if (a % 2 == 0) a ];
echo(list); // ECHO: [2, 4, 6, 8]
```

Note that the *if* element cannot be inside an expression, it should be at the top.

Example

```
// from the input list include all positive odd numbers
// and also all even number divided by 2
list = [-10:5];
echo([for(n=list) if(n%2==0 || n>=0) n%2==0 ? n/2 : n ]); // ECHO: [-5, -4, -3, -2, -1, 0, 1, 1, 3, 2, 5]
// echo([for(n=list) n%2==0 ? n/2 : if(n>=0) n ]); // this would generate a syntactical error
```

Created with the Personal Edition of HelpNDoc: [Full-featured EBook editor](#)

if/else

The if-else construct is equivalent to the conditional expression `?:` except that it can be combined with filter if.

[for (i = list) if (condition(i)) x else y]

When the evaluation of the condition returns true, the expression x is added to the result list else the expression y.

```
// even numbers are halved, positive odd numbers are preserved, negative
odd numbers are eliminated
echo([for (a = [-3:5]) if (a % 2 == 0) [a, a/2] else if (a > 0) [a, a] ]]);
// ECHO: [[-2, -1], [0, 0], [1, 1], [2, 1], [3, 3], [4, 2], [5, 5]];
```

Note that in the expression above the conditional operator could not substitute if-else. It is possible to express this same filter with the conditional operator but with a more cryptic logic:

```
// even numbers are halved, positive odd numbers are preserved, negative
odd numbers are eliminated
echo([for (a = [-3:5]) if (a % 2 == 0 || (a % 2 != 0 && a > 0)) a % 2 ==
0 ? [a, a / 2] : [a, a] ]]);
// ECHO: [[-2, -1], [0, 0], [1, 1], [2, 1], [3, 3], [4, 2], [5, 5]];
```

To bind an else expression to a specific if, it's possible to use parenthesis.

```
// even numbers are dropped, multiples of 4 are substituted by -1
echo([for(i=[0:10]) if(i%2==0) (if(i%4==0) -1 ) else i]);
// ECHO: [-1, 1, 3, -1, 5, 7, -1, 9]
// odd numbers are dropped, multiples of 4 are substituted by -1
echo([for(i=[0:10]) if(i%2==0) if(i%4==0) -1 else i]);
// ECHO: [-1, 2, -1, 6, -1, 10]
```

Created with the Personal Edition of HelpNDoc: [Easily create EPub books](#)

let

The let element allows sequential assignment of variables inside a list comprehension definition.

[for (i = list) let (assignments) a]

Example

```
list = [ for (a = [ 1 : 4 ]) let (b = a*a, c = 2 * b) [ a, b, c ] ];
echo(list); // ECHO: [[1, 1, 2], [2, 4, 8], [3, 9, 18], [4, 16, 32]]
```

Created with the Personal Edition of HelpNDoc: [Write EPub books for the iPad](#)

Nested loops

There are different ways to define nested loops. Defining multiple loop variables inside one for element and multiple for elements produce both flat result lists. To generate nested result lists an additional `[]` markup is required.

```
// nested loop using multiple variables
flat_result1 = [ for (a = [ 0 : 2 ], b = [ 0 : 2 ]) a == b ? 1 : 0 ];
```

```
echo(flat_result1); // ECHO: [1, 0, 0, 0, 1, 0, 0, 0, 1]
```

```
// nested loop using multiple for elements
flat_result2 = [ for (a = [ 0 : 2 ]) for (b = [ 0 : 2 ]) a == b ? 1 : 0 ];
echo(flat_result2); // ECHO: [1, 0, 0, 0, 1, 0, 0, 0, 1]
```

```
// nested loop to generate a bi-dimensional matrix
identity_matrix = [ for (a = [ 0 : 2 ]) [ for (b = [ 0 : 2 ]) a == b ? 1 :
0 ] ];
echo(identity_matrix); // ECHO: [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
```

Created with the Personal Edition of HelpNDoc: [Create iPhone web-based documentation](#)

Advanced Examples

This chapter lists some advanced examples, useful idioms and use-cases for the list comprehension syntax.

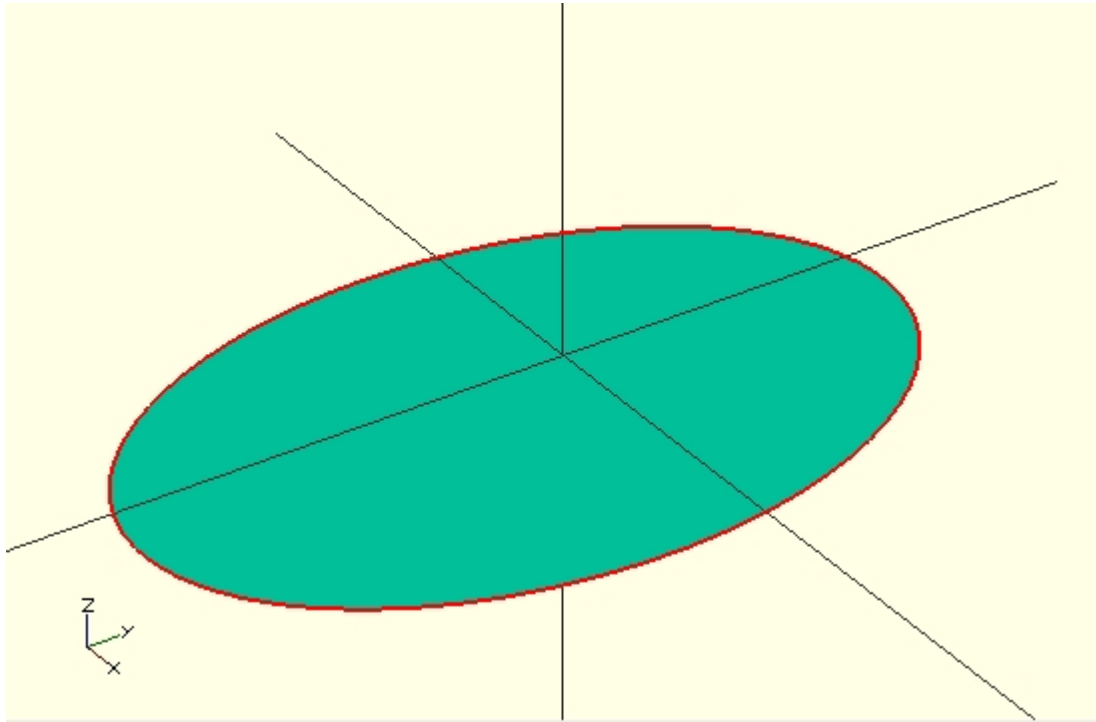
Created with the Personal Edition of HelpNDoc: [Easily create EPub books](#)

Generating vertices for a polygon

Using list comprehension, a parametric equation can be calculated at a number of points to approximate many curves, such as the following example for an ellipse (using `polygon()`):

```
sma = 20; // semi-minor axis
smb = 30; // semi-major axis
polygon(
[ for (a = [0 : 5 : 359]) [ sma * sin(a), smb * cos(a) ] ]
);
```

e advanced examples, useful idioms and use-cases for the list comprehension syntax.



Created with the Personal Edition of HelpNDoc: [Free EPub producer](#)

Flattening a nested vector

List comprehension can be used in a user-defined function to perform tasks on or for vectors. Here is a user-defined function that flattens a nested vector.

```
// input : nested list
// output : list with the outer level nesting removed
function flatten(l) = [ for (a = l) for (b = a) b ] ;
nested_list = [ [ 1, 2, 3 ], [ 4, 5, 6 ] ];
echo(flatten(nested_list)); // ECHO: [1, 2, 3, 4, 5, 6]
```

Created with the Personal Edition of HelpNDoc: [Write eBooks for the Kindle](#)

Sorting a vector

Even a complicated algorithm **Quicksort** becomes doable with `for()`, `if()`, `let()` and [recursion](#):

```
// input : list of numbers
// output : sorted list of numbers
function quicksort(arr) = !(len(arr)>0) ? [] : let(
  pivot = arr[floor(len(arr)/2)],
  lesser = [ for (y = arr) if (y < pivot) y ],
  equal = [ for (y = arr) if (y == pivot) y ],
  greater = [ for (y = arr) if (y > pivot) y ]
) concat(
  quicksort(lesser), equal, quicksort(greater)
);
// use seed in rand() to get reproducible results
unsorted = [for (a = rand(0, 10, 6, 3)) ceil(a)];
echo(unsorted); // ECHO: [6, 1, 8, 9, 3, 2]
echo(quicksort(unsorted)); // ECHO: [1, 2, 3, 6, 8, 9]
```

Selecting elements of a vector

`select()` performs selection and reordering of elements into a new vector.

```
function select(vector,indices) = [ for (index = indices) vector[index] ];
vector1 = [[0,0],[1,1],[2,2],[3,3],[4,4]];
selector1 = [4,0,3];
vector2 = select(vector1,selector1); // [[4, 4], [0, 0], [3, 3]]
vector3 = select(vector1,[0,2,4,4,2,0]); // [[0, 0], [2, 2], [4, 4],[4, 4],
[2, 2], [0, 0]]
// range also works as indices
vector4 = select(vector1, [4:-1:0]); // [[4, 4], [3, 3], [2, 2], [1, 1],
[0, 0]]
```

Concatenating two vectors

Using indices:

```
function cat(L1, L2) = [ for (i=[0:len(L1)+len(L2)-1]) i < len(L1)? L1[i] :
L2[i-len(L1)] ] ;
echo(cat([1,2,3],[4,5])); //concatenates two OpenSCAD lists [1,2,3] and
[4,5], giving [1, 2, 3, 4, 5]
```

Without using indices:

```
function cat(L1, L2) = [ for (L=[L1, L2], a=L) a ];
echo(cat([1,2,3],[4,5])); //concatenates two OpenSCAD lists [1,2,3] and
[4,5], giving [1, 2, 3, 4, 5]
```

Other Language Features

Special Variables

Special variables provide an alternate means of passing arguments to modules and functions. All user, or OpenSCAD, defined variables starting with a '\$' are special variables, similar to special variables in lisp. Modules and function see all outside variables in addition to those passed as arguments or defined internally.

The value for a regular variable is assigned at compile time and is thus static for all calls.

Special variables pass along their value from within the scope ([see scope of variables](#)) from which the module or function is called. This means that special variables can potentially have a different value each time a module or function is called.

```
regular = "regular global"; $special = "special global"; module
```

```

show() echo(" in show ", regular," ", $special ); echo (" outside
", regular," ", $special ); // ECHO: " outside ", "regular
global", " ", "special global" for ( regular = [0:1] ){ echo("in
regular loop ", regular," ", $special ); show();} // ECHO: "in
regular loop ", 0, " ", "special global" // ECHO: " in show ",
"regular global", " ", "special global" // ECHO: "in regular loop
", 1, " ", "special global" // ECHO: " in show ", "regular
global", " ", "special global" for ( $special = [5:6] ){ echo("in
special loop ", regular," ", $special ); show();} // ECHO: "in
special loop ", "regular global", " ", 5 // ECHO: " in show ",
"regular global", " ", 5 // ECHO: "in special loop ", "regular
global", " ", 6 // ECHO: " in show ", "regular global", " ", 6
show(); // ECHO: " in show ", "regular global", " ", "special
global"

```

This is useful when multiple arguments need to be passed thru several layers of module calls.

Several special variables are already defined by OpenSCAD.

Created with the Personal Edition of HelpNDoc: [Full-featured EPub generator](#)

[\\$fa, \\$fs and \\$fn](#)

The \$fa, \$fs and \$fn special variables control the number of facets used to generate an arc:

\$fa is the minimum angle for a fragment. Even a huge circle does not have more fragments than 360 divided by this number. The default value is 12 (i.e. 30 fragments for a full circle). The minimum allowed value is 0.01. Attempting to set a lower value causes a warning.

\$fs is the minimum size of a fragment. The default value is 2 so very small circles have a smaller number of fragments than specified using \$fa. The minimum allowed value is 0.01. Attempting to set a lower value causes a warning.

\$fn is usually the default value of 0. When this variable has a value greater than zero, the other two variables are ignored, and a full circle is rendered using this number of fragments.

The higher the number of fragments, the more memory and CPU consumed; large values can bring many systems to their knees. Depending on the design, \$fn values, and the corresponding results of \$fa & \$fs, should be kept small, at least until the design is finalised when it can be increased for the final result. **A \$fn over 100 is not recommended** or only for specific circumstances, and below 50 would be advisable for performance.

TIP: If you want to create a circle/cylinder/sphere which has an axis aligned integer bounding box (i.e. a bounding box that has integral dimensions, and an integral position) use a value of \$fn that is divisible by 4.

When \$fa and \$fs are used to determine the number of fragments for a circle, then OpenSCAD never uses fewer than 5 fragments.

This is the C code that calculates the number of fragments in a circle:

```

int get_fragments_from_r(double r, double fn, double fs, double
fa) { if (r < GRID_FINE) return 3; if (fn > 0.0) return (int)(fn
>= 3 ? fn : 3); return (int)ceil(fmax(fmin(360.0 / fa, r*2*M_PI /

```

```
fs), 5)); }
```

Or you can embed this OpenSCAD version in your code to work out what's going on, you need to set `r=` to your size

```
echo(n=($fn>0?
($fn>=3?$fn:3):ceil(max(min(360/$fa,r*2*PI/$fs),5))),a_based=360/$
fa,s_based=r*2*PI/$fs);
```

Spheres are first sliced into as many slices as the number of fragments being used to render a circle of the sphere's radius, and then every slice is rendered into as many fragments as are needed for the slice radius. You might have recognized already that the pole of a sphere is usually a pentagon. This is why.

The number of fragments for a cylinder is determined using the greater of the two radii.

The method is also used when rendering circles and arcs from DXF files. The variables have no effect when importing STL files.

You can generate high resolution spheres by resetting the `$fX` values in the instantiating module:

```
$fs = 0.01; sphere(2);
```

or simply by passing the special variable as parameter:

```
sphere(2, $fs = 0.01);
```

You can even scale the special variable instead of resetting it:

```
sphere(2, $fs = $fs * 0.01);
```

Created with the Personal Edition of HelpNDoc: [Easily create HTML Help documents](#)

\$t

The `$t` variable is used for animation. If you enable the animation frame with `view->animate` and give a value for "FPS" and "Steps", the "Time" field shows the current value of `$t`. With this information in mind, you can animate your design. The design is recompiled every `1/"FPS"` seconds with `$t` incremented by `1/"Steps"` for "Steps" times, ending at either `$t=1` or `$t=1-1/steps`.

If "Dump Pictures" is checked, then images are created in the same directory as the .scad file, using the following `$t` values, and with the following naming convention:

Naming Convention

Pattern 1:

- `$t=0/Steps` filename="frame00001.png"
- `$t=1/Steps` filename="frame00002.png"
- `$t=2/Steps` filename="frame00003.png"
- ...
- `$t=1-3/Steps` filename="frame<Steps-2>.png"

- `$t=1-2/Steps filename="frame<Steps-1>.png"`
- `$t=1-1/Steps filename="frame00000.png"`

Or, for other values of Steps, it follows this pattern:

Pattern2:

- `$t=0/Steps filename="frame00001.png"`
- `$t=1/Steps filename="frame00002.png"`
- `$t=2/Steps filename="frame00003.png"`
- ...
- `$t=1-3/Steps filename="frame<Steps-2>.png"`
- `$t=1-2/Steps filename="frame<Steps-1>.png"`
- `$t=1-1/Steps filename="frame<Steps-0>.png"`
- `$t=1-0/Steps filename="frame00000.png"`

Which pattern it chooses appears to be an unpredictable, but consistent determined by Steps. For example, when Steps=4, it follows the first pattern, and outputs a total of 4 files. When Steps=3, it follows the second pattern, and also outputs 4 files. It always outputs either Steps or Steps+1 files, though it may not be predictable which. When finished, it wraps around and recreate each of the files, looping through and recreating them forever.

Created with the Personal Edition of HelpNDoc: [Produce online help for Qt applications](#)

`$vpr`, `$vpt` and `$vpd`

These contain the current viewport rotation and translation and camera distance - at the time of doing the rendering. Moving the viewport does not update them. During an animation they are updated for each frame.

- `$vpr` shows rotation
- `$vpt` shows translation (i.e. won't be affected by rotate and zoom)
- `$vpd` shows the camera distance [**Note:** Requires version **2015.03**]

Example

```
cube([10, 10, $vpr[0] / 10]);
```

which makes the cube change size based on the view angle, if an animation loop is active (which does not need to use the `$t` variable)

You can also make bits of a complex model vanish as you change the view.

All three variables are writable, but only assignments at the top-level of the main file has an effect on the viewport. [**Note:** Requires version **2015.03**]

Example

```
$vpr = [0, 0, $t * 360];
```

which allows a simple 360 degree rotation around the Z axis in animation mode.

The menu command *Edit - Paste Viewport Rotation/Translation* copies the current value of the viewport, but not the current `$vpr` or `$vpt`.

\$preview

\$preview is true, when in OpenCSG preview (F5). \$preview is false, when in render (F6).

This can, for example, be used to reduce detail during preview to save time, without losing detail in the final rendered result:

```
$fn = $preview ? 12 : 72; sphere(r = 1);
```

Note that the render module does not affect \$preview:

```
render(){ $fn = $preview ? 12 : 72; sphere(r = 1); }
```

Another use could be to make the preview show an assembly view and the render generate just the printed parts laid out for printing.

If printed parts need extra features that are removed post printing, for example support for suspended holes, then the preview can omit these to show the finished part after post processing.

When OpenSCAD is run from the command line \$preview is only true when generating a PNG image with OpenCSG. It is false when generating STL, DXF and SVG files with CGAL. It is also false when generating CSG and ECHO files. This may or may not be what you want, but you can always override it on the command line like any other variable with the -D option.

Echo Statements

This function prints the contents to the compilation window (aka Console). Useful for debugging code. Also see the String function [str\(\)](#).

Numeric values are rounded to 5 significant digits.

It can be handy to use 'variable=variable' as the expression to easily label the variables, see the example below.

Usage examples[\[edit\]](#)

Usage examples:

```
my_h=50; my_r=100; echo("This is a cylinder with h=", my_h, " and  
r=", my_r); echo(my_h=my_h, my_r=my_r); // shortcut  
cylinder(h=my_h, r=my_r);
```

Shows in the Console as

```
ECHO: "This is a cylinder with h=", 50, " and r=", 100 ECHO: my_h  
= 50, my_r = 100
```

Rounding examples[\[edit\]](#)

An example for the rounding:

```
a=1.0;
b=1.000002;
echo(a);
echo(b);
if(a==b){ //while echoed the same, the values are still distinct
echo ("a==b");
}else if(a>b){
echo ("a>b");
}else if(a<b){
echo ("a<b");
}else{
echo ("???");
}
```

Small and large Numbers[\[edit\]](#)

```
c=10000002;
d=0.0000002;
echo(c); //1e+06
echo(d); //2e-06
```

HTML[\[edit\]](#)

HTML output is not officially supported, however depending on the OpenSCAD version, some HTML tags were rendered in the console window.

Created with the Personal Edition of HelpNDoc: [Produce online help for Qt applications](#)

Echo Function

Echo can be used in expression context to print information while the function/expression is evaluated. The output is generated before the expression evaluation to allow debugging of recursive functions.

Example

```
a = 3; b = 5; // echo() prints values before evaluating the
expression r1 = echo(a, b) a * b; // ECHO: 3, 5 // using let it's
still easy to output the result r2 = let(r = 2 * a * b) echo(r) r;
// ECHO: 30 // use echo statement for showing results echo(r1,
r2); // ECHO: 15, 30
```

A more complex example shows how echo() can be used in both descending and ascending path of a recursive function. The result() helper function is a simple way to output the value of an expression after evaluation.

Example printing both input values and result of recursive sum()

```
v = [4, 7, 9, 12]; function result(x) = echo(result = x) x;
function sum(x, i = 0) = echo(str("x[", i, "]= ", x[i]))
result(len(x) > i ? x[i] + sum(x, i + 1) : 0); echo("sum(v) = ",
sum(v)); // ECHO: "x[0]=4" // ECHO: "x[1]=7" // ECHO: "x[2]=9" //
ECHO: "x[3]=12" // ECHO: "x[4]=undef" // ECHO: result = 0 // ECHO:
```

```
result = 12 // ECHO: result = 21 // ECHO: result = 28 // ECHO:
result = 32 // ECHO: "sum(v) = ", 32
```

Created with the Personal Edition of HelpNDoc: [Produce online help for Qt applications](#)

Render

Forces the generation of a mesh even in preview mode. Useful when the boolean operations become too slow to track.

Needs description.

Usage examples:

```
render(convexity = 2) difference() { cube([20, 20, 150], center =
true); translate([-10, -10, 0]) cylinder(h = 80, r = 10, center =
true); translate([-10, -10, +40]) sphere(r = 10); translate([-10,
-10, -40]) sphere(r = 10); }
```

Created with the Personal Edition of HelpNDoc: [Create cross-platform Qt Help files](#)

Surface

Surface reads [Heightmap](#) information from text or image files.

Parameters

file

String. The path to the file containing the heightmap data.

center

Boolean. This determines the positioning of the generated object. If true, object is centered in X- and Y-axis. Otherwise, the object is placed in the positive quadrant. Defaults to false.

invert

Boolean. Inverts how the color values of imported images are translated into height values. This has no effect when importing text data files. Defaults to false. **[Note: Requires version 2015.03]**

convexity

Integer. The convexity parameter specifies the maximum number of front sides (back sides) a ray intersecting the object might penetrate. This parameter is only needed for correctly displaying the object in OpenCSG preview mode and has no effect on the final rendering.

Text file format[\[edit\]](#)

The format for text based heightmaps is a matrix of numbers that represent the height for a specific point. Rows are mapped to the Y-axis, columns to the X axis. The numbers must be separated by spaces or tabs. Empty lines and lines starting with a # character are ignored.

Images[\[edit\]](#)

[Note: Requires version 2015.03]

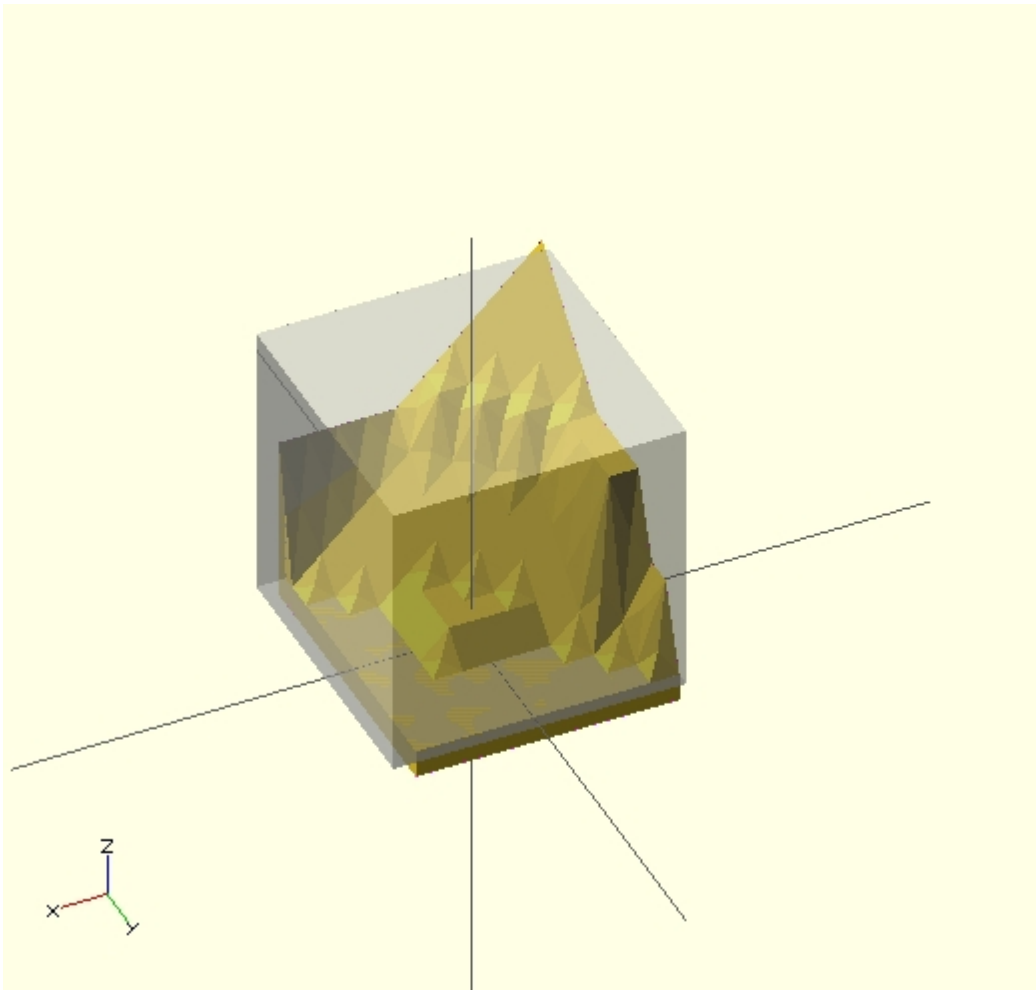
Currently only PNG images are supported. Alpha channel information of the image is ignored and the height for the pixel is determined by converting the color value to [Grayscale](#) using the linear luminance for the sRGB color space ($Y = 0.2126R + 0.7152G + 0.0722B$). The gray scale values are scaled to be in the range 0 to 100.

Examples[\[edit\]](#)

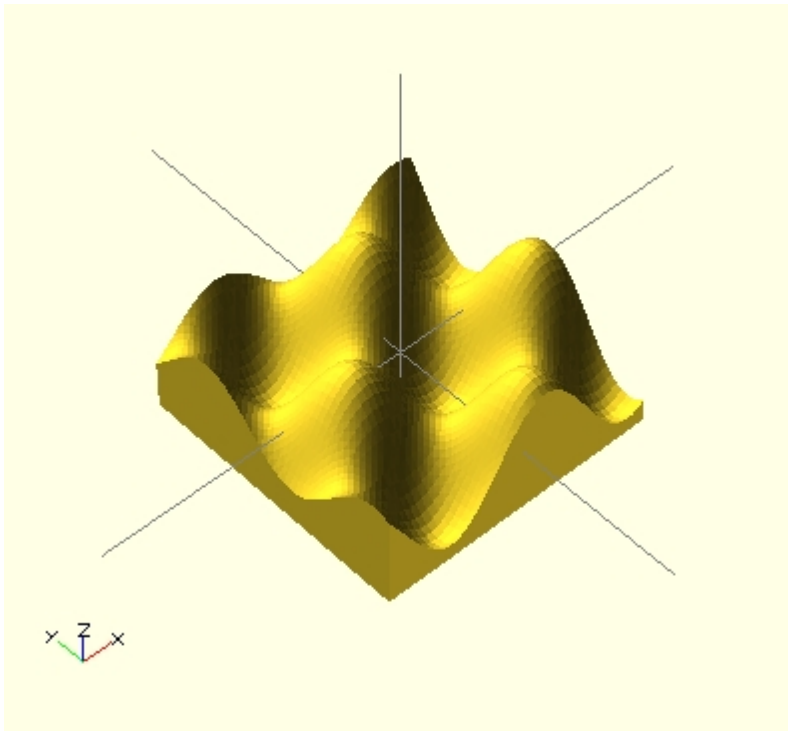
Example 1:

```
//surface.scad surface(file = "surface.dat", center = true,
convexity = 5); %translate([0,0,5])cube([10,10,10], center =true);
```

```
#surface.dat 10 9 8 7 6 5 5 5 5 5 9 8 7 6 6 4 3 2 1 0 8 7 6 6 4 3
2 1 0 0 7 6 6 4 3 2 1 0 0 0 6 6 4 3 2 1 1 0 0 0 6 6 3 2 1 1 1 0 0
0 6 6 2 1 1 1 1 0 0 0 6 6 1 0 0 0 0 0 0 0 3 1 0 0 0 0 0 0 0 3 0
0 0 0 0 0 0 0 0
```

Result:**Example 2**

```
// example010.dat generated using octave: // d = (sin(1:0.2:10))' *
cos(1:0.2:10)) * 10; // save("example010.dat", "d");
intersection() { surface(file = "example010.dat", center = true,
convexity = 5); rotate(45, [0, 0, 1]) surface(file =
"example010.dat", center = true, convexity = 5); }
```

**Example 3:**

[Note: Requires version **2015.03**]

```
// Example 3a scale([1, 1, 0.1]) surface(file = "smiley.png",  
center = true);
```

```
// Example 3b scale([1, 1, 0.1]) surface(file = "smiley.png",  
center = true, invert = true);
```

Example 3: Using surface() with a PNG image as heightmap input.



Input image



Example 3a: surface(invert = false)



Example 3b: surface (invert = true)

Example 4:

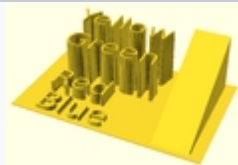
```
// Example 4 surface(file = "BRGY-Grey.png", center = true, invert  
= false);
```

•



PNG Test File

•



3D Surface

Created with the Personal Edition of HelpNDoc: [Free HTML Help documentation generator](#)

Search

Created with the Personal Edition of HelpNDoc: [Easy CHM and documentation editor](#)

Search Usage

Created with the Personal Edition of HelpNDoc: [Full-featured Help generator](#)

Search Arguments

Created with the Personal Edition of HelpNDoc: [Create cross-platform Qt Help files](#)

Search Usage Examples

The `search()` function is a general-purpose function to find one or more (or all) occurrences of a value or list of values in a vector, string or more complex list-of-list construct.

Search Usage[\[edit\]](#)

```
search( match_value , string_or_vector [, num_returns_per_match [, index_col_num ] ] );
```

Search Arguments[\[edit\]](#)

- **match_value**

Can be a single string value. Search loops over the characters in the string and searches for each one in the second argument. The second argument must be a string or a list of lists (this second case is not recommended). The search function does **not** search for substrings.

- Can be a single numerical value.
- Can be a list of values. The search function searches for each item on the list.
- To search for a list or a full string give the list or string as a single element list such as `["abc"]` to search for the string "abc" (See **Example 9**) or `[[6,7,8]]` to search for the list `[6,7,8]`. Without the extra brackets, search looks separately for each item in the list.
- If `match_value` is boolean then search returns `undef`.

- **string_or_vector**

The string or vector to search for matches.

- If **match_value** is a string then this should be a string and the string is searched for individual character matches to the characters in **match_value**
- If this is a list of lists, `v=[[a0,a1,a2...],[b0,b1,b2,...],[c0,c1,c2,...],...]` then search looks only at one index position of the sublists. By default this is position 0, so the search looks only at `a0`, `b0`, `c0`, etc. The **index_col_num** parameter changes which index is searched.
- If **match_value** is a string and this parameter is a list of lists then the characters of the string are tested against the appropriate index entry in the list of lists. However, if any characters fail to find a match a warning message is printed and that return value is excluded from the output (if **num_returns_per_match** is 1). This means that the length of the output is unpredictable.

- **num_returns_per_match** (default: 1)

By default, search only looks for one match per element of `match_value` to return as a list of indices

- If `num_returns_per_match > 1`, search returns a list of lists of up to `num_returns_per_match` index values for each element of `match_value`.

See **Example 8** below.

- If `num_returns_per_match = 0`, search returns a list of lists of **all** matching index values for each element of `match_value`.

See **Example 6** below.

- **index_col_num** (default: 0)

As noted above, when searching a list of lists, search looks only at one index position of each sublist. That index position is specified by **index_col_num**.

- See **Example 5** below for a simple usage example.

Search Usage Examples[\[edit\]](#)

See **example023.scad** included with OpenSCAD for a renderable example.

Index values return as list[\[edit\]](#)

Example	Code	Result
---------	------	--------

1	<code>search("a", "abcdabcd");</code>	<code>[0]</code>
2	<code>search("e", "abcdabcd");</code>	<code>[]</code>
3	<code>search("a", "abcdabcd", 0);</code>	<code>[[0,4]]</code>
4	<pre>data= [["a",1],["b",2], ["c",3],["d",4],["a",5], ["b",6],["c",7],["d",8], ["e",9]]; search("a", data, num_returns_per_match= 0);</pre>	<code>[[0,4]]</code> (see also Example 6 below)

Search on different column; return Index values[\[edit\]](#)

Example 5:

```
data= [ ["a",1],["b",2],["c",3],["d",4],["a",5],["b",6],["c",7],
["d",8],["e",3] ]; echo(search(3, data)); // Searches index 0, so
it doesn't find anything echo(search(3, data,
num_returns_per_match=0, index_col_num=1));
```

Outputs:

```
ECHO: [] ECHO: [2, 8]
```

Search on list of values[\[edit\]](#)

Example 6: Return all matches per search vector element.

```
data= [ ["a",1],["b",2],["c",3],["d",4],["a",5],["b",6],["c",7],
["d",8],["e",9] ]; search("abc", data, num_returns_per_match=0);
```

Returns:

```
[[0,4],[1,5],[2,6]]
```

Example 7: Return first match per search vector element; special case return vector.

```
data= [ ["a",1],["b",2],["c",3],["d",4],["a",5],["b",6],["c",7],
["d",8],["e",9] ]; search("abc", data, num_returns_per_match=1);
```

Returns:

```
[0,1,2]
```

Example 8: Return first two matches per search vector element; vector of vectors.

```
data= [ ["a",1],["b",2],["c",3],["d",4],["a",5],["b",6],["c",7],
```

```
[ "d",8],[ "e",9] ]; search("abce", data, num_returns_per_match=2);
```

Returns:

```
[[0,4],[1,5],[2,6],[8]]
```

Search on list of strings[\[edit\]](#)

Example 9:

```
lTable2=[ [ "cat",1],[ "b",2],[ "c",3],[ "dog",4],[ "a",5],[ "b",6],
[ "c",7],[ "d",8],[ "e",9],[ "apple",10],[ "a",11] ];
lSearch2=[ "b", "zzz", "a", "c", "apple", "dog"];
l2=search(lSearch2,lTable2); echo(str("Default list string search
(",lSearch2,"): ",l2));
```

Returns

```
ECHO: "Default list string search ([ "b", "zzz", "a", "c", "apple",
"dog"]): [1, [], 4, 2, 9, 3]"
```

Getting the right results[\[edit\]](#)

```
// workout which vectors get the results v=[ [ "O",2],[ "p",3],
[ "e",9],[ "n",4],[ "S",5],[ "C",6],[ "A",7],[ "D",8] ]; //
echo(v[0]); // -> [ "O",2] echo(v[1]); // -> [ "p",3] echo(v[1]
[0],v[1][1]); // -> "p",3 echo(search("p",v)); // find "p" -> [1]
echo(search("p",v)[0]); // -> 1 echo(search(9,v,0,1)); // find 9 -
-> [2] echo(v[search(9,v,0,1)[0]]); // -> [ "e",9]
echo(v[search(9,v,0,1)[0]][0]); // -> "e" echo(v[search(9,v,0,1)
[0]][1]); // -> 9 echo(v[search("p",v,1,0)[0]][1]); // -> 3
echo(v[search("p",v,1,0)[0]][0]); // -> "p"
echo(v[search("d",v,1,0)[0]][0]); // "d" not found -> undef
echo(v[search("D",v,1,0)[0]][1]); // -> 8
```

Created with the Personal Edition of HelpNDoc: [Full-featured EBook editor](#)

OpenSCAD Version

version() and version_num() returns the OpenSCAD version number.

- The version() function returns the OpenSCAD version as a vector, e.g. [2011, 09, 23]
- The version_num() function returns the OpenSCAD version as a number, e.g. 20110923

Created with the Personal Edition of HelpNDoc: [Create help files for the Qt Help Framework](#)

parent_module(n) and \$parent_modules

\$parent_modules contains the number of modules in the instantiation stack. parent_module(i) returns the name of the module i levels above the current module in the instantiation stack. The stack is independent of where the modules are defined. It's where they're instantiated that counts. This can, for example, be used to build a BOM (Bill Of Material).

Example:

```
module top() { children(); } module middle() { children(); } top()
middle() echo(parent_module(0)); // prints "middle" top() middle()
echo(parent_module(1)); // prints "top"
```

Created with the Personal Edition of HelpNDoc: [Easily create Qt Help files](#)

assert

see also *Assertion (software development)*

Assert evaluates a logical expression. If the expression evaluates to false, the generation of the preview/render is stopped, and an error condition is reported via the console. The report consists of a string representation of the expression and an additional string (optional) that is specified in the assert command.

```
assert(condition); assert(condition, string);
```

Parameters

condition

Expression. The expression to be evaluated as check for the assertion.

message

String. Optional message to be output in case the assertion failed.

Example[\[edit\]](#)

The simplest example is a simple `assert(false);`, e.g. in a file named `assert_example1.scad`.

```
cube();
assert(false);
sphere();
// ERROR: Assertion 'false' failed in file assert_example1.scad, line 2
```

This example has little use, but the simple `assert(false);` can be used in code sections that should be unreachable.

Checking parameters[\[edit\]](#)

A useful example is checking the validity of input parameters:

```
module row(cnt = 3){
// Count has to be a positive integer greater 0
assert(cnt > 0);
for (i = [1 : cnt]) {
translate([i * 2, 0, 0]) sphere();
}
}
row(0);
// ERROR: Assertion '(cnt > 0)' failed in file assert_example2.scad, line
3
```

Adding message[\[edit\]](#)

When writing a library, it could be useful to output additional information to the user in case of an

failed assertion.

```
module row(cnt = 3){
  assert(cnt > 0, "Count has to be a positive integer greater 0");
  for(i = [1 : cnt]) {
    translate([i * 2, 0, 0]) sphere();
  }
}
row(0);
// ERROR: Assertion '(cnt > 0)': "Count has to be a positive integer
greater 0" failed in file assert_example3.scad, line 2
```

Using assertions in function[\[edit\]](#)

Assert returns its children, so when using it in a function you can write

```
function f(a, b) =
  assert(a < 0, "wrong a") // assert input
  assert(b > 0, "wrong b") // assert input
  let (c = a + b) // derive a new value from input
  assert(c != 0, "wrong c") // assert derived value
  a * b; // calculate
```

Created with the Personal Edition of HelpNDoc: [Easily create iPhone documentation](#)

User Defined Functions and Modules

Created with the Personal Edition of HelpNDoc: [Produce Kindle eBooks easily](#)

Introduction

Created with the Personal Edition of HelpNDoc: [Create help files for the Qt Help Framework](#)

Scope

Created with the Personal Edition of HelpNDoc: [Free iPhone documentation generator](#)

Functions

Created with the Personal Edition of HelpNDoc: [Full-featured Kindle eBooks generator](#)

Recursive Functions

Created with the Personal Edition of HelpNDoc: [Qt Help documentation made easy](#)

Function Literals

Created with the Personal Edition of HelpNDoc: [Generate EPub eBooks with ease](#)

Modules

Created with the Personal Edition of HelpNDoc: [Easy to use tool to create HTML Help files and Help web sites](#)

Object modules

Created with the Personal Edition of HelpNDoc: [Benefits of a Help Authoring Tool](#)

Operator Modules

Created with the Personal Edition of HelpNDoc: [Easily create PDF Help documents](#)

Children

Created with the Personal Edition of HelpNDoc: [Free help authoring environment](#)

Further Module Examples

Created with the Personal Edition of HelpNDoc: [Easily create PDF Help documents](#)

Recursive Modules

Created with the Personal Edition of HelpNDoc: [Create iPhone web-based documentation](#)

Overwriting built-in modules

Created with the Personal Edition of HelpNDoc: [Easily create Help documents](#)

Overwriting built functions

Created with the Personal Edition of HelpNDoc: [Create help files for the Qt Help Framework](#)

Debugging Aids

Created with the Personal Edition of HelpNDoc: [Free Qt Help documentation generator](#)

Advanced Concept

Created with the Personal Edition of HelpNDoc: [Free HTML Help documentation generator](#)

Background Modifier

Created with the Personal Edition of HelpNDoc: [Create cross-platform Qt Help files](#)

Debug Modifier

Created with the Personal Edition of HelpNDoc: [Benefits of a Help Authoring Tool](#)

Root Modifier

Created with the Personal Edition of HelpNDoc: [Produce online help for Qt applications](#)

Disable Modifier

Created with the Personal Edition of HelpNDoc: [Free Qt Help documentation generator](#)

Echo Statements

Created with the Personal Edition of HelpNDoc: [Free Web Help generator](#)

External Libraries and code files

Created with the Personal Edition of HelpNDoc: [Easily create Qt Help files](#)

Use and Include

Created with the Personal Edition of HelpNDoc: [Free help authoring environment](#)

Directory separators

Created with the Personal Edition of HelpNDoc: [Easy to use tool to create HTML Help files and Help web sites](#)

Variables

Created with the Personal Edition of HelpNDoc: [Free Kindle producer](#)

Scope of Variables

Created with the Personal Edition of HelpNDoc: [Full-featured Kindle eBooks generator](#)

Overwriting variables

Created with the Personal Edition of HelpNDoc: [What is a Help Authoring tool?](#)

Example "Ring-Library"

Created with the Personal Edition of HelpNDoc: [Easily create Help documents](#)

Nested Include and Use

Created with the Personal Edition of HelpNDoc: [Create HTML Help, DOC, PDF and print manuals from 1 single source](#)

Import

Created with the Personal Edition of HelpNDoc: [Create iPhone web-based documentation](#)

Parameters

Created with the Personal Edition of HelpNDoc: [Single source CHM, PDF, DOC and HTML Help creation](#)

Convexity

Created with the Personal Edition of HelpNDoc: [Create help files for the Qt Help Framework](#)

Notes

Created with the Personal Edition of HelpNDoc: [Free EPub and documentation generator](#)

Import DXF

Created with the Personal Edition of HelpNDoc: [Create help files for the Qt Help Framework](#)

Import STL

Created with the Personal Edition of HelpNDoc: [Create cross-platform Qt Help files](#)

surface

Created with the Personal Edition of HelpNDoc: [Produce Kindle eBooks easily](#)

Parameters

Created with the Personal Edition of HelpNDoc: [Easily create eBooks](#)

Text file Format

Created with the Personal Edition of HelpNDoc: [Free help authoring tool](#)

Images

Created with the Personal Edition of HelpNDoc: [Full-featured multi-format Help generator](#)

Examples

Created with the Personal Edition of HelpNDoc: [Easily create iPhone documentation](#)

MCAD Library

MCAD Library

This library contains components commonly used in designing and mocking up mechanical designs. It is currently unfinished and you can expect some API changes, however many things are already working.

This library was created by various authors as named in the individual files' comments. All the files are licensed under the LGPL 2.1 (see <http://creativecommons.org/licenses/LGPL/2.1/> or the included file lgpl-2.1.txt), some of them allow distribution under more permissive terms (as described in the files' comments).

Usage

You can import these files in your scripts with use <MCAD/filename.scad>, where 'filename' is one of the files listed below like 'motors' or 'servos'. Some files include useful constants which will be available with include <MCAD/filename.scad>, which should be safe to use on all included files (ie. no top level code should create geometry).

(There is a bug/feature that prevents including constants from files that "include" other files - see the openscad mailing list archives for more details. Since the maintainers aren't very responsive, may have to work around this somehow)

If you host your project in git, you can do `git submodule add URL PATH` in your repo to import this library as a git submodule for easy usage. Then you need to do a `git submodule update --init` after cloning. When you want to update the submodule, do `cd PATH; git checkout master; git pull`. See `git help submodule` for more info.

Currently Provided Tools:

- `regular_shapes.scad`
 - regular polygons, ie. 2D
 - regular polyhedrons, ie. 3D
- `involute_gears.scad` (<http://www.thingiverse.com/thing:3575>):
 - `gear()`
 - `bevel_gear()`
 - `bevel_gear_pair()`
- `gears.scad` (Old version):
 - `gear(number_of_teeth, circular_pitch OR diametrial_pitch, pressure_angle OPTIONAL, clearance OPTIONAL)`
- `motors.scad`:
 - `stepper_motor_mount(nema_standard, slide_distance OPTIONAL, mochup OPTIONAL)`

Tools (alpha and beta quality):

- `nuts_and_bolts.scad`: for creating metric and imperial bolt/nut holes
- `bearing.scad`: standard/custom bearings
- `screw.scad`: screws and augers
- `materials.scad`: color definitions for different materials
- `stepper.scad`: NEMA standard stepper outlines
- `servos.scad`: servo outlines
- `boxes.scad`: box with rounded corners
- `triangles.scad`: simple triangles
- `3d_triangle.scad`: more advanced triangles

Very generally useful functions and constants:

- `math.scad`: general math functions
- `constants.scad`: mathematical constants
- `curves.scad`: mathematical functions defining curves
- `units.scad`: easy metric units
- `utilities.scad`: geometric functions and misc. useful stuff
- `teardrop.scad` (<http://www.thingiverse.com/thing:3457>): parametric teardrop module
- `shapes.scad`: DEPRECATED simple shapes by Catarina Mota
- `polyholes.scad`: holes that should come out well when printed

Other:

- `alphabet_block.scad`
- `bitmap.scad`
- `letter_necklace.scad`
- `name_tag.scad`
- `height_map.scad`
- `trochoids.scad`
- `libtriangles.scad`
- `layouts.scad`
- `transformations.scad`

2Dshapes.scad
 gridbeam.scad
 fonts.scad
 unregular_shapes.scad
 metric_fasteners.scad
 lego_compatibility.scad
 multiply.scad
 hardware.scad

External utils that generate and process openscad code:

openscad_testing.py: testing code, see below
 openscad_utils.py: code for scraping function names etc.

Created with the Personal Edition of HelpNDoc: [Free Kindle producer](#)

regular_shapes.scad

Created with the Personal Edition of HelpNDoc: [Produce electronic books easily](#)

2D regular shapes

Created with the Personal Edition of HelpNDoc: [Create iPhone web-based documentation](#)

regular_polygon(sides, radius)

Created with the Personal Edition of HelpNDoc: [Produce Kindle eBooks easily](#)

n-gons 2D shapes

Created with the Personal Edition of HelpNDoc: [Easily create EPub books](#)

triangle(radius)

Created with the Personal Edition of HelpNDoc: [Free CHM Help documentation generator](#)

pentagon(radius)

Created with the Personal Edition of HelpNDoc: [Create help files for the Qt Help Framework](#)

hexagon(radius)

Created with the Personal Edition of HelpNDoc: [Easy EBook and documentation generator](#)

heptagon(radius)

Created with the Personal Edition of HelpNDoc: [Easily create CHM Help documents](#)

octagon(radius)

Created with the Personal Edition of HelpNDoc: [Create HTML Help, DOC, PDF and print manuals from 1 single source](#)

nonagon(radius)

Created with the Personal Edition of HelpNDoc: [Free HTML Help documentation generator](#)

decagon(radius)

Created with the Personal Edition of HelpNDoc: [News and information about help authoring tools and software](#)

hendecagon(radius)

Created with the Personal Edition of HelpNDoc: [Free HTML Help documentation generator](#)

dodecagon(radius)

Created with the Personal Edition of HelpNDoc: [Easy EBook and documentation generator](#)

ring(inside_diameter, thickness)

Created with the Personal Edition of HelpNDoc: [Free HTML Help documentation generator](#)

ellipse(width, height)

Created with the Personal Edition of HelpNDoc: [Full-featured Help generator](#)

egg_outline(width, thickness)

Created with the Personal Edition of HelpNDoc: [Easily create Qt Help files](#)

3D regular shapes

Created with the Personal Edition of HelpNDoc: [Easy CHM and documentation editor](#)

cone

Created with the Personal Edition of HelpNDoc: [Produce electronic books easily](#)

oval_prism

Created with the Personal Edition of HelpNDoc: [Produce Kindle eBooks easily](#)

oval_tube

Created with the Personal Edition of HelpNDoc: [Produce online help for Qt applications](#)

cylinder_tube

Created with the Personal Edition of HelpNDoc: [Create help files for the Qt Help Framework](#)

triangle_prism

Created with the Personal Edition of HelpNDoc: [Create help files for the Qt Help Framework](#)

triangle_tube

Created with the Personal Edition of HelpNDoc: [Create iPhone web-based documentation](#)

pentagon_prism

Created with the Personal Edition of HelpNDoc: [Easily create iPhone documentation](#)

pentagon_tube

Created with the Personal Edition of HelpNDoc: [Full-featured Documentation generator](#)

hexagon_prism

Created with the Personal Edition of HelpNDoc: [Free help authoring tool](#)

hexagon_tube

Created with the Personal Edition of HelpNDoc: [Easily create EPub books](#)

heptagon_prism

Created with the Personal Edition of HelpNDoc: [Easily create CHM Help documents](#)

heptagon_tube

Created with the Personal Edition of HelpNDoc: [Free CHM Help documentation generator](#)

octagon_prism

Created with the Personal Edition of HelpNDoc: [Benefits of a Help Authoring Tool](#)

octagon_tube

Created with the Personal Edition of HelpNDoc: [Create HTML Help, DOC, PDF and print manuals from 1 single source](#)

nonagon_prism

Created with the Personal Edition of HelpNDoc: [Full-featured multi-format Help generator](#)

decagon_prism

Created with the Personal Edition of HelpNDoc: [Free iPhone documentation generator](#)

hendecagon_prism

Created with the Personal Edition of HelpNDoc: [Easy to use tool to create HTML Help files and Help web sites](#)

dodecagon_prism

Created with the Personal Edition of HelpNDoc: [Free HTML Help documentation generator](#)

torus

Created with the Personal Edition of HelpNDoc: [Produce electronic books easily](#)

torus2

Created with the Personal Edition of HelpNDoc: [Easy EBook and documentation generator](#)

oval_torus

Created with the Personal Edition of HelpNDoc: [Write eBooks for the Kindle](#)

triangle_pyramid

Created with the Personal Edition of HelpNDoc: [Create help files for the Qt Help Framework](#)

square_pyramid

Created with the Personal Edition of HelpNDoc: [iPhone web sites made easy](#)

egg

Created with the Personal Edition of HelpNDoc: [Free iPhone documentation generator](#)

involute_gears.scad

Created with the Personal Edition of HelpNDoc: [Free iPhone documentation generator](#)

bevel_gear_pair()

Created with the Personal Edition of HelpNDoc: [Create cross-platform Qt Help files](#)

bevel_gear()

Created with the Personal Edition of HelpNDoc: [Produce online help for Qt applications](#)

gear()

Created with the Personal Edition of HelpNDoc: [Free PDF documentation generator](#)

Tests

Created with the Personal Edition of HelpNDoc: [News and information about help authoring tools and software](#)

test_gears()

Created with the Personal Edition of HelpNDoc: [Easily create iPhone documentation](#)

`test_meshing_double_helix()`

Created with the Personal Edition of HelpNDoc: [Full-featured EBook editor](#)

`test_bevel_gear()`

Created with the Personal Edition of HelpNDoc: [Single source CHM, PDF, DOC and HTML Help creation](#)

`test_bevel_gear_pair()`

Created with the Personal Edition of HelpNDoc: [Free Web Help generator](#)

`test_backlash()`

Created with the Personal Edition of HelpNDoc: [News and information about help authoring tools and software](#)

dotSCAD Library

Created with the Personal Edition of HelpNDoc: [Qt Help documentation made easy](#)

2D modules

Created with the Personal Edition of HelpNDoc: [Free EBook and documentation generator](#)

arc

Created with the Personal Edition of HelpNDoc: [Easily create EBooks](#)

pie

Created with the Personal Edition of HelpNDoc: [Free PDF documentation generator](#)

rounded_square

Created with the Personal Edition of HelpNDoc: [Easy EBook and documentation generator](#)

line2d

Created with the Personal Edition of HelpNDoc: [Free EPub producer](#)

polyline2d

Created with the Personal Edition of HelpNDoc: [Easy EPub and documentation editor](#)

hull_polyline2d

Created with the Personal Edition of HelpNDoc: [Produce online help for Qt applications](#)

hexagons

Created with the Personal Edition of HelpNDoc: [Easily create EBooks](#)

polytransversals

Created with the Personal Edition of HelpNDoc: [Produce electronic books easily](#)

multi_line_text

Created with the Personal Edition of HelpNDoc: [Easy EBook and documentation generator](#)

voronoi2d

Created with the Personal Edition of HelpNDoc: [Qt Help documentation made easy](#)

3D modules

Created with the Personal Edition of HelpNDoc: [Create iPhone web-based documentation](#)

rounded_cube

Created with the Personal Edition of HelpNDoc: [Easy EPub and documentation editor](#)

rounded_cylinder

Created with the Personal Edition of HelpNDoc: [Easy EPub and documentation editor](#)

crystal_ball

Created with the Personal Edition of HelpNDoc: [Easily create EPub books](#)

line3d

Created with the Personal Edition of HelpNDoc: [Create help files for the Qt Help Framework](#)

polyline3d

Created with the Personal Edition of HelpNDoc: [Easy EBook and documentation generator](#)

hull_polyline3d

Created with the Personal Edition of HelpNDoc: [iPhone web sites made easy](#)

function_grapher

Created with the Personal Edition of HelpNDoc: [Free help authoring environment](#)

sweep

Created with the Personal Edition of HelpNDoc: [Free EBook and documentation generator](#)

loft

Created with the Personal Edition of HelpNDoc: [Full-featured Kindle eBooks generator](#)

starburst

Created with the Personal Edition of HelpNDoc: [Create help files for the Qt Help Framework](#)

voronoi3d

Created with the Personal Edition of HelpNDoc: [Produce Kindle eBooks easily](#)

Transformations

Created with the Personal Edition of HelpNDoc: [Easy EPub and documentation editor](#)

along_width

Created with the Personal Edition of HelpNDoc: [Free Qt Help documentation generator](#)

hollow_out

Created with the Personal Edition of HelpNDoc: [Create cross-platform Qt Help files](#)

bend

Created with the Personal Edition of HelpNDoc: [Free Kindle producer](#)

shear

Created with the Personal Edition of HelpNDoc: [Produce Kindle eBooks easily](#)

2D functions

Created with the Personal Edition of HelpNDoc: [Easily create PDF Help documents](#)

in_shape

Created with the Personal Edition of HelpNDoc: [Create iPhone web-based documentation](#)

bijection_offset

Created with the Personal Edition of HelpNDoc: [Full-featured Documentation generator](#)

trim_shape

Created with the Personal Edition of HelpNDoc: [Create iPhone web-based documentation](#)

triangulate

Created with the Personal Edition of HelpNDoc: [Generate EPub eBooks with ease](#)

contours

Created with the Personal Edition of HelpNDoc: [Generate Kindle eBooks with ease](#)

2D/3D functions

Created with the Personal Edition of HelpNDoc: [Easily create HTML Help documents](#)

cross_sections

Created with the Personal Edition of HelpNDoc: [Free Qt Help documentation generator](#)

paths2sections

Created with the Personal Edition of HelpNDoc: [Easily create eBooks](#)

path_scaling_sections

Created with the Personal Edition of HelpNDoc: [Free iPhone documentation generator](#)

bezier_surface

Created with the Personal Edition of HelpNDoc: [Free CHM Help documentation generator](#)

bezier_smooth

Created with the Personal Edition of HelpNDoc: [Free CHM Help documentation generator](#)

midpt_smooth

Created with the Personal Edition of HelpNDoc: [Easy CHM and documentation editor](#)

in_polyline

Created with the Personal Edition of HelpNDoc: [Produce electronic books easily](#)

Paths

Created with the Personal Edition of HelpNDoc: [Create help files for the Qt Help Framework](#)

arc_path

Created with the Personal Edition of HelpNDoc: [Free help authoring tool](#)

bspline_curve

Created with the Personal Edition of HelpNDoc: [Produce online help for Qt applications](#)

bezier_curve

Created with the Personal Edition of HelpNDoc: [Free help authoring environment](#)

helix

Created with the Personal Edition of HelpNDoc: [Easily create EBooks](#)

golden_spiral

Created with the Personal Edition of HelpNDoc: [Qt Help documentation made easy](#)

archimedean_spiral

Created with the Personal Edition of HelpNDoc: [Easily create Help documents](#)

sphere_spiral

Created with the Personal Edition of HelpNDoc: [Easy CHM and documentation editor](#)

torus_knot

Created with the Personal Edition of HelpNDoc: [Benefits of a Help Authoring Tool](#)

Extrusion

Created with the Personal Edition of HelpNDoc: [Easily create Help documents](#)

box_extrude

Created with the Personal Edition of HelpNDoc: [Create cross-platform Qt Help files](#)

ellipse_extrude

Created with the Personal Edition of HelpNDoc: [Easy to use tool to create HTML Help files and Help web sites](#)

stereographic_extrude

Created with the Personal Edition of HelpNDoc: [Free help authoring environment](#)

rounded_extrude

Created with the Personal Edition of HelpNDoc: [Produce online help for Qt applications](#)

bend_extrude

Created with the Personal Edition of HelpNDoc: [Free Qt Help documentation generator](#)

2D Shape

Created with the Personal Edition of HelpNDoc: [Easily create Help documents](#)

shape_taiwan

Created with the Personal Edition of HelpNDoc: [Produce electronic books easily](#)

shape_arc

Created with the Personal Edition of HelpNDoc: [Free PDF documentation generator](#)

shape_pie

Created with the Personal Edition of HelpNDoc: [Generate Kindle eBooks with ease](#)

shape_circle

Created with the Personal Edition of HelpNDoc: [Qt Help documentation made easy](#)

shape_ellipse

Created with the Personal Edition of HelpNDoc: [Single source CHM, PDF, DOC and HTML Help creation](#)

shape_square

Created with the Personal Edition of HelpNDoc: [Free help authoring tool](#)

shape_trapezium

Created with the Personal Edition of HelpNDoc: [Create HTML Help, DOC, PDF and print manuals from 1 single source](#)

shape_cyclicpolygon

Created with the Personal Edition of HelpNDoc: [Create help files for the Qt Help Framework](#)

shape_pentagram

Created with the Personal Edition of HelpNDoc: [Generate Kindle eBooks with ease](#)

shape_starburst

Created with the Personal Edition of HelpNDoc: [Easy EBook and documentation generator](#)

shape_superformula

Created with the Personal Edition of HelpNDoc: [Generate EPub eBooks with ease](#)

shape_glue2circles

Created with the Personal Edition of HelpNDoc: [Free Web Help generator](#)

shape_path_extend

Created with the Personal Edition of HelpNDoc: [Generate EPub eBooks with ease](#)

2D Shape extrusions

Created with the Personal Edition of HelpNDoc: [Produce electronic books easily](#)

path_extrude

Created with the Personal Edition of HelpNDoc: [Single source CHM, PDF, DOC and HTML Help creation](#)

ring_extrude

Created with the Personal Edition of HelpNDoc: [Easily create EPub books](#)

helix_extrude

Created with the Personal Edition of HelpNDoc: [Produce online help for Qt applications](#)

golden_spiral_extrude

Created with the Personal Edition of HelpNDoc: [Free HTML Help documentation generator](#)

archimedean_spiral_extrude

Created with the Personal Edition of HelpNDoc: [Free help authoring environment](#)

sphere_spiral_extrude

Created with the Personal Edition of HelpNDoc: [Free EPub producer](#)

Utils

Created with the Personal Edition of HelpNDoc: [Full-featured Help generator](#)

util/sub_str

Created with the Personal Edition of HelpNDoc: [iPhone web sites made easy](#)

util/split_str

Created with the Personal Edition of HelpNDoc: [Easily create EBooks](#)

util/parse_number

Created with the Personal Edition of HelpNDoc: [Produce electronic books easily](#)

util/reverse

Created with the Personal Edition of HelpNDoc: [Generate EPub eBooks with ease](#)

util/slice

Created with the Personal Edition of HelpNDoc: [Free EPub producer](#)

util/sort

Created with the Personal Edition of HelpNDoc: [Create HTML Help, DOC, PDF and print manuals from 1 single source](#)

util/rand

Created with the Personal Edition of HelpNDoc: [Free EPub producer](#)

util/fibseq

Created with the Personal Edition of HelpNDoc: [Free Qt Help documentation generator](#)

util/bsearch

Created with the Personal Edition of HelpNDoc: [Create help files for the Qt Help Framework](#)

util/has

Created with the Personal Edition of HelpNDoc: [Full-featured Help generator](#)

util/dedup

Created with the Personal Edition of HelpNDoc: [Write eBooks for the Kindle](#)

util/flat

Created with the Personal Edition of HelpNDoc: [Create help files for the Qt Help Framework](#)

Matrix

Created with the Personal Edition of HelpNDoc: [What is a Help Authoring tool?](#)

matrix/m_cumulate

Created with the Personal Edition of HelpNDoc: [Create help files for the Qt Help Framework](#)

matrix/m_translation

Created with the Personal Edition of HelpNDoc: [Create help files for the Qt Help Framework](#)

matrix/m_rotation

Created with the Personal Edition of HelpNDoc: [Produce online help for Qt applications](#)

matrix/m_scaling

Created with the Personal Edition of HelpNDoc: [Easily create Web Help sites](#)

matrix/m_mirror

Created with the Personal Edition of HelpNDoc: [Easy EBook and documentation generator](#)

matrix/m_shearing

Created with the Personal Edition of HelpNDoc: [Easily create Help documents](#)

Point Transformation

Created with the Personal Edition of HelpNDoc: [Full-featured Help generator](#)

ptf/ptf_rotate

Created with the Personal Edition of HelpNDoc: [Full-featured EPub generator](#)

ptf/ptf_x_twist

Created with the Personal Edition of HelpNDoc: [Produce Kindle eBooks easily](#)

ptf/ptf_y_twist

Created with the Personal Edition of HelpNDoc: [Produce Kindle eBooks easily](#)

ptf/ptf_circle

Created with the Personal Edition of HelpNDoc: [Create HTML Help, DOC, PDF and print manuals from 1 single source](#)

ptf/ptf_bend

Created with the Personal Edition of HelpNDoc: [Write EPub books for the iPad](#)

ptf/ptf_ring

Created with the Personal Edition of HelpNDoc: [Free HTML Help documentation generator](#)

ptf/ptf_sphere

Created with the Personal Edition of HelpNDoc: [Write EPub books for the iPad](#)

ptf/ptf_torus

Created with the Personal Edition of HelpNDoc: [Easy EBook and documentation generator](#)

Turtle

Created with the Personal Edition of HelpNDoc: [Easily create EPub books](#)

turtle/turtle2d

Created with the Personal Edition of HelpNDoc: [Single source CHM, PDF, DOC and HTML Help creation](#)

turtle/turtle3d

Created with the Personal Edition of HelpNDoc: [Free CHM Help documentation generator](#)

turtle/t2d

Created with the Personal Edition of HelpNDoc: [Full-featured Help generator](#)

turtle/t3d

Created with the Personal Edition of HelpNDoc: [Full-featured EBook editor](#)

Pixel

Created with the Personal Edition of HelpNDoc: [Generate Kindle eBooks with ease](#)

pixel/px_line

Created with the Personal Edition of HelpNDoc: [Easy EPub and documentation editor](#)

pixel/px_polyline

Created with the Personal Edition of HelpNDoc: [Full-featured EBook editor](#)

pixel/px_circle

Created with the Personal Edition of HelpNDoc: [Free EBook and documentation generator](#)

pixel/px_cylinder

Created with the Personal Edition of HelpNDoc: [Free CHM Help documentation generator](#)

pixel/px_sphere

Created with the Personal Edition of HelpNDoc: [Easily create iPhone documentation](#)

pixel/px_polygon

Created with the Personal Edition of HelpNDoc: [News and information about help authoring tools and software](#)

pixel/px_from

Created with the Personal Edition of HelpNDoc: [Free iPhone documentation generator](#)

pixel/px_ascii

Created with the Personal Edition of HelpNDoc: [Easily create Web Help sites](#)

pixel/px_gray

Created with the Personal Edition of HelpNDoc: [What is a Help Authoring tool?](#)

Part

Created with the Personal Edition of HelpNDoc: [Benefits of a Help Authoring Tool](#)

part/connector_jpg

Created with the Personal Edition of HelpNDoc: [Write EPub books for the iPad](#)

part/cone

Created with the Personal Edition of HelpNDoc: [Generate EPub eBooks with ease](#)

part/join_T

Created with the Personal Edition of HelpNDoc: [Easily create EBooks](#)

Surface

Created with the Personal Edition of HelpNDoc: [Full-featured EPub generator](#)

surface/sf_square

Created with the Personal Edition of HelpNDoc: [Free HTML Help documentation generator](#)

surface/sf_bend

Created with the Personal Edition of HelpNDoc: [Free Qt Help documentation generator](#)

surface/sf_ring

Created with the Personal Edition of HelpNDoc: [Free EBook and documentation generator](#)

surface/sf_sphere

Created with the Personal Edition of HelpNDoc: [Free Web Help generator](#)

surface/sf_torus

Created with the Personal Edition of HelpNDoc: [Produce Kindle eBooks easily](#)

surface/sf_solidity

Created with the Personal Edition of HelpNDoc: [Create cross-platform Qt Help files](#)

Noise

Created with the Personal Edition of HelpNDoc: [Free HTML Help documentation generator](#)

noise/nz_perlin1

Created with the Personal Edition of HelpNDoc: [Easy to use tool to create HTML Help files and Help web sites](#)

noise/nz_perlin1s

Created with the Personal Edition of HelpNDoc: [Produce online help for Qt applications](#)

noise/nz_perlin2

Created with the Personal Edition of HelpNDoc: [Easily create CHM Help documents](#)

noise/nz_perlin2s

Created with the Personal Edition of HelpNDoc: [Easily create EBooks](#)

noise/nz_perlin3

Created with the Personal Edition of HelpNDoc: [Generate Kindle eBooks with ease](#)

noise/nz_perlin3s

Created with the Personal Edition of HelpNDoc: [Easy CHM and documentation editor](#)

noise/nz_worley2

Created with the Personal Edition of HelpNDoc: [Easily create HTML Help documents](#)

noise/nz_worley2s

Created with the Personal Edition of HelpNDoc: [Free HTML Help documentation generator](#)

noise/nz_worley3

Created with the Personal Edition of HelpNDoc: [Free help authoring environment](#)

noise/nz_worley3s

Created with the Personal Edition of HelpNDoc: [Free CHM Help documentation generator](#)

noise/nz_cell

Created with the Personal Edition of HelpNDoc: [News and information about help authoring tools and software](#)

BOSL Library

Created with the Personal Edition of HelpNDoc: [Generate Kindle eBooks with ease](#)

Commonly Used

Commonly Used

The most commonly used transformations, manipulations, and shortcuts.

Created with the Personal Edition of HelpNDoc: [Free PDF documentation generator](#)

transforms.scad

Created with the Personal Edition of HelpNDoc: [Easy CHM and documentation editor](#)

Translations

Created with the Personal Edition of HelpNDoc: [Easy to use tool to create HTML Help files and Help web sites](#)

move()

move()

use<transforms.scad>

Usage:

- move([x], [y], [z]) ...
- move([x, y, z]) ...

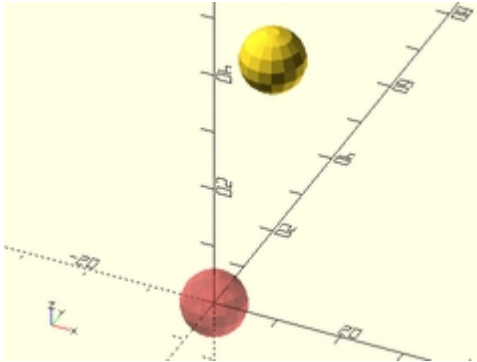
Description: Moves/translated children.

Argument	What it does
x	X axis translation.
y	Y axis translation.

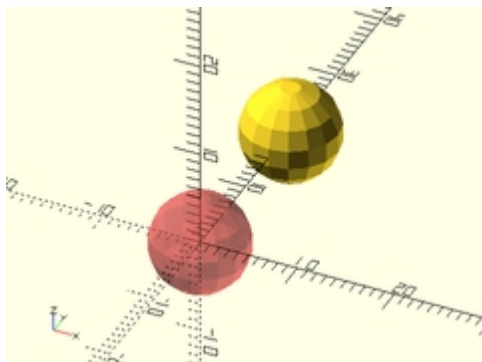
z	Z axis translation.
---	---------------------

Example 1:

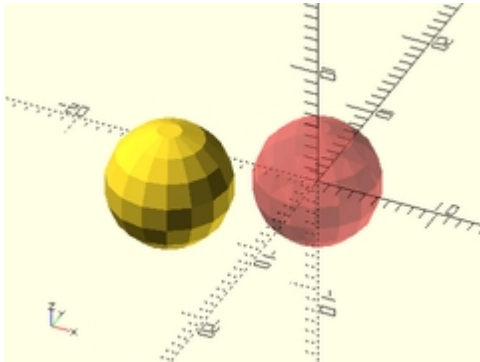
```
#sphere(d = 10);  
move([0, 20, 30]) sphere(d = 10);
```

**Example 2:**

```
#sphere(d = 10);  
move(y = 20) sphere(d = 10);
```

**Example 3:**

```
#sphere(d = 10);  
move(x = -10, y = -5) sphere(d = 10);
```



Created with the Personal Edition of HelpNDoc: [Easily create Help documents](#)

xmove()

xmove()

use<transforms.scad>

Usage:

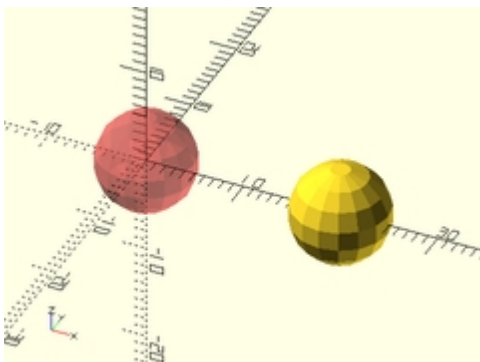
- xmove(x) ...

Description: Moves/translates children the given amount along the X axis.

Argument	What it does
x	Amount to move right along the X axis. Negative values move left.

Example:

```
#sphere(d = 10);
xmove(20) sphere(d = 10);
```



Created with the Personal Edition of HelpNDoc: [Easy to use tool to create HTML Help files and Help web sites](#)

`ymove()`

Created with the Personal Edition of HelpNDoc: [Easy CHM and documentation editor](#)

`zmove()`

Created with the Personal Edition of HelpNDoc: [News and information about help authoring tools and software](#)

`left()`

Created with the Personal Edition of HelpNDoc: [Easily create iPhone documentation](#)

`right()`

Created with the Personal Edition of HelpNDoc: [Free PDF documentation generator](#)

`fwd() / fordware()`

Created with the Personal Edition of HelpNDoc: [Easily create Web Help sites](#)

`back()`

Created with the Personal Edition of HelpNDoc: [Full-featured Kindle eBooks generator](#)

`down()`

Created with the Personal Edition of HelpNDoc: [Easy to use tool to create HTML Help files and Help web sites](#)

`up()`

Created with the Personal Edition of HelpNDoc: [Easily create Qt Help files](#)

shapes.scad

Created with the Personal Edition of HelpNDoc: [Free EPub and documentation generator](#)

masks.scad

Created with the Personal Edition of HelpNDoc: [Free HTML Help documentation generator](#)

threading.scad

Created with the Personal Edition of HelpNDoc: [Single source CHM, PDF, DOC and HTML Help creation](#)

paths.scad

Created with the Personal Edition of HelpNDoc: [Easily create Qt Help files](#)

beziers.scad

Created with the Personal Edition of HelpNDoc: [Easy EBook and documentation generator](#)

Standard Parts

Created with the Personal Edition of HelpNDoc: [Produce electronic books easily](#)

involute_gears.scad

Created with the Personal Edition of HelpNDoc: [Produce electronic books easily](#)

joiners.scad

Created with the Personal Edition of HelpNDoc: [Easily create Qt Help files](#)

sliders.scad

Created with the Personal Edition of HelpNDoc: [Easily create Help documents](#)

metric_screws.scad

Created with the Personal Edition of HelpNDoc: [Create cross-platform Qt Help files](#)

linear_bearings.scad

Created with the Personal Edition of HelpNDoc: [Full-featured Help generator](#)

nema_steppers.scad

Created with the Personal Edition of HelpNDoc: [Free PDF documentation generator](#)

phillips_drive.scad

Created with the Personal Edition of HelpNDoc: [Produce online help for Qt applications](#)

torx_drive.scad

Created with the Personal Edition of HelpNDoc: [Benefits of a Help Authoring Tool](#)

wiring.scad

Created with the Personal Edition of HelpNDoc: [Easily create iPhone documentation](#)

Miscellaneous

Created with the Personal Edition of HelpNDoc: [Full-featured multi-format Help generator](#)

constants.scad

Created with the Personal Edition of HelpNDoc: [Free Qt Help documentation generator](#)

math.scad

Created with the Personal Edition of HelpNDoc: [Easily create Qt Help files](#)

convex_hull.scad

Created with the Personal Edition of HelpNDoc: [Easily create EPub books](#)

quaternions.scad

Created with the Personal Edition of HelpNDoc: [Full-featured Kindle eBooks generator](#)

triangulation.scad

Created with the Personal Edition of HelpNDoc: [Create iPhone web-based documentation](#)

debug.scad

Created with the Personal Edition of HelpNDoc: [Full-featured EBook editor](#)

Nut Job Library

Created with the Personal Edition of HelpNDoc: [Easily create Web Help sites](#)
