OpenSCAD User Manual/Print version

Table of Contents

Introduction

OpenSCAD is a software for creating solid 3D CAD objects.
It is free software (http://www.gnu.org/philosophy/free-sw.html) and available for GNU/Linux (http://www.gnu.org/), MS Windows and Apple OS X.

Unlike most free software for creating 3D models (such as the well-known application Blender (http://www.blender.org/)), 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 product, is not an interactive modeller. 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 and enables you to easily change any step in the modelling process or 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 computers GPU, but is an approximation of the model and can produce artifacts; Preview uses OpenCSG (http://opencsg.org/) 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) or 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 http://openscad.org/. You may find extra information in the mailing list (http://rocklinux.net/mailman/listinfo/openscad).

People who don't want to (or can't) install new software on their computer may be able to use OpenJSCAD (http://OpenJSCAD.org/), a port of OpenSCAD that runs in a web browser, if your browser supports WebGL

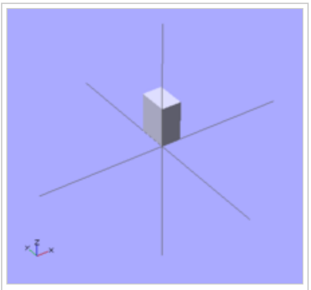
A pt_BR translation of this document is avaliable on GitHub repository (not completed/on development) [1] (http://www.github.com/ubb3rsith/OpenSCAD_doc_ptBR)

First Steps

For our first model we will create a simple 2 x 3 x 4 cuboid. In the openSCAD editor, type the following one line command:

Usage example 1 - simple cuboid:

```
cube([2,3,4]);
```


OpenSCAD Simple Cuboid

Compiling and rendering our first model

The cuboid can now be compiled and rendered by pressing F6 or F5 while the openSCAD editor has focus. cube([5,29.1,6]);

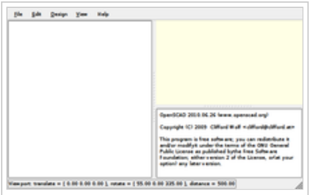
See also

Positioning an object

Open one of the many examples that come with OpenSCAD (*File*, *Examples*). Or you can copy and paste this simple example into the OpenSCAD window:

Usage example 1

```
difference() {
    cube(30, center=true);
    sphere(20);
}
translate([0, 0, 30]) {
    cylinder(h=40, r=10);
}
```



```
OpenSCAD 2010.06.26 (www.openscad.org)

Copyright (C) 2009 Clifford Wolf <clifford@clifford.at>

This program is free software; you can redistribute it
and/or modify it under the terms of the GNU General
Public License as published by the Free Software
Foundation; either version 2 of the License, or (at your
option) any later version.
```

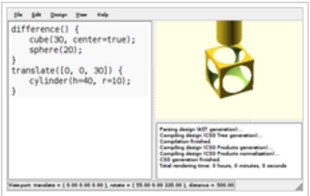
```
Viewport: translate = [ 0.00 0.00 0.00 ], rotate = [ 55.00 0.00 325.00 ], distance = 500.00
```

OpenSCAD after starting

Then **press F5** to get a graphical preview of what you typed (or **press F6** to get a graphical view).

You get three types of movement in the preview frame:

1. Drag with left mouse button to rotate the view. The bottom line will change the rotate values.
2. Drag with an other mouse button (or control-drag under OSX) to translate (move) the view. The bottom line will change translate values.
3. Use the mouse scroll to zoom in and out. Alternatively you can use the + and - keys, or right-drag with the mouse while pressing a shift key (or control-shift-drag under OSX). The Viewport line at the bottom of the window will show a change in the distance value.



```
difference() {
    cube(30, center=true);
    sphere(20);
}
translate([0, 0, 30]) {
    cylinder(h=40, r=10);
}
```

```
Parsing design (AST generation)...
Compiling design (CSG Tree generation)...
Compilation finished.
Compiling design (CSG Products generation)...
Compiling design (CSG Products normalization)...
CSG generation finished.
Total rendering time: 0 hours, 0 minutes, 0 seconds
```

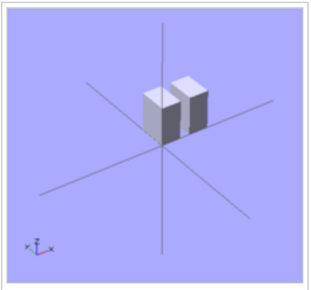
```
Viewport: translate = [ 0.00 0.00 0.00 ], rotate = [ 55.00 0.00 325.00 ], distance = 500.00
```

OpenSCAD after pasting the example code and pressing F5

We have already seen how to create a simple cuboid. Our next task is to attempt to use the translate positioning command to place an identical cuboid next to the existing cuboid:

Usage example 1 - positioning an object:

```
cube([2,3,4]);
translate([3,0,0]) {
  cube([2,3,4]);
}
```

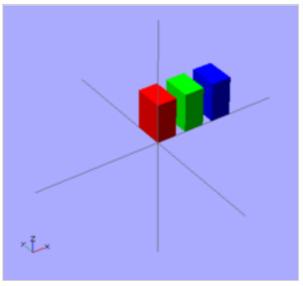

OpenSCAD positioning an object

There is no semicolon following the translate command

Notice that there is no semicolon following the translate command. This is because the translate command relates to the following object. If the semicolon was not omitted, then the effect of the position translation would end, and the second cuboid would be placed at the same position as the first cuboid. We can change the color of an object by giving it RGB values. Instead of the traditional RGB values from 0 to 255 floating point values are used from 0.0 to 1.0. Note! Changing the colors only works in Preview mode (F5). Render mode (F6) does not currently support color.

Usage example 1 - changing the color of an object:

```
color([1,0,0]) cube([2,3,4]);
translate([3,0,0])
color([0,1,0]) cube([2,3,4]);
translate([6,0,0])
color([0,0,1]) cube([2,3,4]);
```


OpenSCAD changing the color of an object

Color names can be used in the 2011.12 version (and newer). The names are the same used for Web colors (http://en.wikipedia.org/wiki/Web_colors). For example: `color("red") cube();`

If you think of the entire command as a sentence, then `color()` is an "adjective" that describes the "object" of the sentence (which is a "noun"). In this case, the object is the `cube()` to be created. The adjective is placed before the noun in the sentence, like so: `color() cube();`. In the same way, `translate()` can be thought of as a "verb" that acts upon the object, and is placed like this: `translate() color() cube();`. The following code will produce the same result:

```
translate([6,0,0])
{
   color([0,0,1])    // notice that there is NO semicolon
   cube([2,3,4]);    // notice the semicolon is at the end of all related commands
}
```

The openscad model view window provides a variety of view options.

CGAL Surfaces

The surface view is the initial model view that appears when the model code is first rendered.

CGAL Grid Only

The Grid Only view presents only the "scaffolding" beneath the surface, also known as a wireframe. Think of the Eiffel Tower.

A wire frame is a visual presentation of a three dimensional or physical object. Using a wire frame model allows visualization of the underlying design structure of a 3D model. Since wireframe renderings are relatively simple and fast to calculate, they are often used in cases where a high screen frame rate is needed (for instance, when working with a particularly complex 3D model, or in real-time systems that model exterior phenomena). When greater graphical detail is desired, surface textures can be added automatically after completion of the initial rendering of the wireframe. This allows the designer to quickly review changes or rotate the object to new desired views without long delays associated with more realistic rendering. The wire frame format is also well suited and widely used in programming tool paths for DNC (Direct Numerical Control) machine tools. Wireframe models are also used as the input for CAM (computer-aided manufacturing). Wireframe is the most abstract and least realistic of the three main CAD models. This method of modelling consists only of lines, points and curves defining the edges of an object. (From Wikipedia: http://en.wikipedia.org/wiki/Wire-frame_model)

The OpenCSG View

This view mode utilizes the open constructive solid geometry library to generate the model view utilizing OpenGL. If the OpenCSG library is not available or the video card or drivers do not support OpenGL, then this view will produce no visible output.

The thrown together view

The thrown together view provides all the previous views, in the same screen.

The OpenSCAD User Interface

User Interface

The user interface of OpenSCAD has three parts

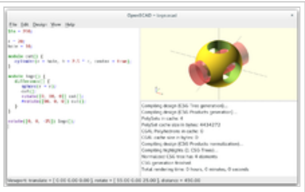
- The viewing area
- The console window
- The text editor

Viewing area

Preview and rendering output goes into the viewing area. Using the *Show Axes* menu entry an indicator for the coordinate axes can be enabled.

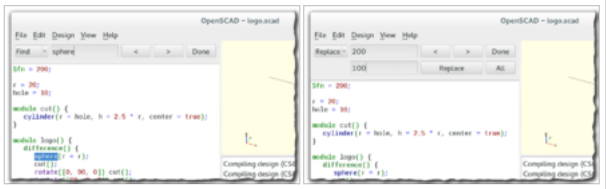
Console window

Status information, warnings and errors are displayed in the console window.


Main Window of OpenSCAD with a small program generating the OpenSCAD-Logo.

Text editor

The built-in text editor provides basic editing features like text search & replace and also supports syntax highlighting. There are predefined color schemes which can be selected in the Preferences dialog.



```
$fn = 200;

r = 20;
hole = 10;

module cut() {
  cylinder(r = hole, h = 2.5 * r, center = true);
}

module logo() {
  difference() {
    sphere(r = r);
    cut();
    rotate([0, 90, 0]) cut();
```

OpenSCAD Editor with Find / Replace functionality.

View navigation

The viewing area is navigated primarily using the mouse:

- Dragging with the left mouse button rotates the view along the axes of the viewing area. It preserves the vertical axis' direction.
- Dragging with the left mouse button when the shift key is pressed rotates the view along the vertical axis and the axis pointing towards the user.
- Dragging with the right mouse button moves the viewing area.
- For zooming, there are four ways:
  - using the scroll wheel
  - dragging with the middle mouse button
  - dragging with the right or middle mouse button and the shift key pressed
  - the keys + and -

Rotation can be reset using the shortcut Ctrl+0. Movement can be reset using the shortcut Ctrl+P.

View setup

The viewing area can be configured to use different rendering methods and other options using the View menu. Most of the options described here are available using shortcuts as well.

Render modes

OpenCSG (F9)

This method produces instantaneous results, but has low frame rates when working with highly nonconvex objects.

Note that selecting the OpenCSG mode using F9 will switch to the last generated OpenCSG view, but will not re-evaluate the source code. You may want to use the *Compile* function (F5, found in the *Design* menu) to re-evaluate the source code, build the OpenCSG objects and *then* switch to OpenCSG view.

Implementation Details

In OpenCSG mode, the OpenCSG library (http://opencsg.org/) is used for generating the visible model. This library uses advanced OpenGL features (2.0) like the Z buffer and does not require an explicit description of the resulting mesh – instead, it tracks how objects are to be combined. For example, when rendering a spherical dent in a cube, it will first render the cube on the graphics card and then render the sphere, but instead of using the Z buffer to **hide** the parts of the sphere that are covered by the cube, it will render **only** those parts of the sphere, visually resulting in a cube with a spherical dent.

CGAL (Surfaces and Grid, F10 and F11)

This method might need some time when first used with a new program, but will then have higher framerates.

As before with OpenCSG, F10 and F11 only enable CGAL display mode and don't update the underlying objects; for that, use the *Compile and Render* function (F6, found in the *Design* menu).

To combine the benefits of those two display methods, you can selectively wrap parts of your program in a render function and force them to be baken into a mesh even with OpenCSG mode enabled.

Implementation Details

The acronym CGAL refers to The Open Source Computational Geometry Algorithms Library.

In CGAL mode, the CGAL library is used to compute the mesh of the root object, which is then displayed using simple OpenGL.

View options

Show Edges (Ctrl+1)

If *Show Edges* is enabled, both OpenCSG and CGAL mode will render edges as well as faces, CGAL will even show vertices. In CGAL grid mode, this option has no effect.

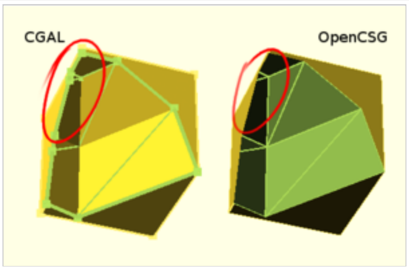
Enabling this option shows the difference between OpenCSG and CGAL quite clearly: While in CGAL mode you see an edge drawn everywhere it "belongs", OpenCSG will not show edges resulting from boolean operations – this is because they were never explicitly calculated but are just where one object's Z clipping begins or ends.

Show Axes **(Ctrl+2)**

If *Show Axes* is enabled, the origin of the global coordinate system will be indicated by an orthogonal axes indicator. Additionally, a smaller axes indicator with axes names will be shown in the lower left corner of the viewing area. The smaller axes indicator is marked x, y, z and coloured red, green, blue respectively.

Show Crosshairs **(Ctrl+3)**

If *Show Crosshairs* is enabled, the center of the viewport will be indicated by four lines pointing in the room diagonal directions of the global coordinate system. This is useful when aligning the viewing area to a particular point in the model to keep it centered on screen during rotation.



The difference between the CGAL and OpenCSG approaches can be seen at edges created by boolean operations.

Animation

The *Animate* option adds an animation bar to the lower edge of the screen. As soon as *FPS* and *Steps* are set (reasonable values to begin with are 10 and 100, respectively), the current *Time* is incremented by 1/*Steps*, *FPS* times per second, until it reaches 1, when it wraps back to 0.

Every time *Time* is changed, the program is re-evaluated with the variable `$t` set to the current time. Read more about how `$t` is used in section Other_Language_Features

View alignment

The menu items *Top, Bottom, …, Diagonal* and *Center* (Ctrl+4, Ctrl+5, …, Ctrl+0, Ctrl+P) align the view to the global coordinate system.

Top, Bottom, Left, Right, Front and *Back* align it in parallel to the axes, the *Diagonal* option aligns it diagonally as it is aligned when OpenSCAD starts.

The *Center* option will put the coordinate center in the middle of the screen (but not rotate the view).

By default, the view is in *Perspective* mode, meaning that distances far away from the viewer will look shorter, as it is common with eyes or cameras. When the view mode is changed to *Orthogonal*, visible distances will not depend on the camera distance (the view will simulate a camera in infinite distance with infinite focal length). This is especially useful in combination with the *Top* etc. options described above, as this will result in a 2D image similar to what one would see in an engineering drawing.

The OpenSCAD Language

Introduction

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

Comments

Comments are a way of leaving notes within the 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 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 additional to decimal numerals, the following names for special numbers are defined:

- PI

OpenSCAD has only a single kind of number, which is an 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 behaviour of numbers in mathematics:

- 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 will be -0 if the result is negative, otherwise it will be 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 equal.

Note that '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);
```

Note that '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.

Boolean Values

Boolean values are truth values. There are two Boolean values, named 'true' and 'false'. A Boolean value is passed as the 'center' argument to the 'cube' and 'cylinder' primitives, as the argument to 'if', and as the arguments to the logical operators '!' (not), '&&' (and), and '||' (or).

In all of these contexts, you can actually pass any type of value. Most values, when used in a Boolean context, are equivalent to 'true', but there are a few that are equivalent to 'false'. The values that count as false are:

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

Note that nan (Not A Number) counts as true.

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

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

- \" → "
- \\ → \
- \t → tab
- \n → newline
- \r → carriage return

Note: This behavior is new since OpenSCAD-2011.04. You can upgrade old files using the following sed command: `sed 's/\\/\\\\/' non-escaped.scad > escaped.scad`

Example:

```
echo("The quick brown fox \tjumps \"over\" the lazy dog.\rThe quick brown fox.\nThe \\lazy\\ dog.");
```

Output:
```
ECHO: "The quick brown fox jumps "over" the lazy dog.
The quick brown fox.
The \lazy\ dog."
```

Output: in OpenSCAD version 2013.02.28
```
ECHO: "The quick brown fox \tjumps \"over\" the lazy dog.
The quick brown fox.\nThe \\lazy\\ dog."
```

Vectors

A vector is a sequence of zero or more OpenSCAD values. Vectors are most commonly used to represent points in 3-space (as [x,y,z] triples), to represent lists of points, and to represent the size of a cuboid (also as an [x,y,z] triple).

A vector is written as a list of zero or more expressions, separated by commas, and enclosed in square brackets, Eg, [] or [10,20,30].

Example

```
deck = [64, 89, 18];
cube(deck);
```

Output A cube with the sizes: X = 64, Y = 89, Z = 18.

Vectors selection

You can also refer to individual values in a vector with vector[number]. number starts from 0.

Example

```
deck = [64, 89, 18];
translate([0,0,deck[2]]) cube(deck);
```

Output The same cube as the previous example would be raised by 18 on the Z axis, since vector indices are numbered [0,1,2] for [X,Y,Z] respectively.

Matrix

A matrix is a vector of vectors.

Example

```
mr = [
      [cos(angle), -sin(angle)],
      [sin(angle),  cos(angle)]
     ];
```

Output Define a 2D rotation matrix.

Ranges

Ranges define a series of numerical values.

Range: [<*start*>:<*end*>] - iterate from *start* to *end* inclusive with a fixed increment of 1. Both *start* and *end* can be fractions. It also works if *end* is smaller than *start* but as of Version 2014.03 this usage is deprecated and will produce a warning message.

Range: [<*start*>:<*increment*>:<*end*>] - iterate from *start* to *end* with the given *increment*. The increment can be a fraction.
It's valid to use a negative increment, in this case *end* must be smaller than (or equal to) *start*.

Warning: If the increment is not an even divider of <*end*>-<*start*>, the iterator value for the last iteration will be <*end*>-(<*end*>-<*start*> mod <*increment*>).

Note for Version < 2014.03: The increment is given as an absolute value and cannot be negative. If <*end*> is smaller than <*start*> the increment should remain unchanged.

Example

```
r = [0.5 : 2.5];
for (n = r) echo(n);
```

Output
ECHO: 0.5
ECHO: 1.5
ECHO: 2.5

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.

Note that numeric operations may also return nan 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.

Variables

Variables in OpenSCAD are simply an identifier followed by an assignment via an expression (i.e. identifier = expression).

Example:

```
myvar = 5 + 4;
```

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' below. 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 variables value to be changed for this purpose.

The variable retains its last assigned value at compile time, in line with Functional programming languages. Unlike Imperative languages, such a C, OpenSCAD is not an iterative language, as such the concept of *x = x + 1* is not valid, get to understand this concept and you will understand the beauty of OpenSCAD.

Currently it's not possible to do assignments at any place (the only places are file top-level and module top-level). So it is invalid inside an *if/else* or *for* loop, if you need it inside the *for* loop you need to use the assign() module.

Update: [***Note:*** *Requires version* ***2015.03***]

Since OpenSCAD-2015.03, the restriction on where assignments can be made have been lifted: Variables can now be assigned in any scope. Note that assignments are only valid in the scope in which they are defined - you are still not allowed to leak values to an outer scope.

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

Note! Anonymous scopes are not considered scopes:

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

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

Output Variable a is undef Variable a is tested undefined

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

Because OpenSCAD calculates its variable values at compile-time, not run-time, the last variable assignment will apply everywhere the variable is used (with some exceptions, mentioned below). It may be helpful to think of them as override-able constants rather than as variables.

Example:

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

    a = 5;
    echo(a);
```

Output

```
ECHO: 5
ECHO: 5
```

This also means that you can not reassign a variable inside an *if* block:

Example:

```
a=0;
if (a==0)
  {
  a=1; // <- this line will generate an error.
  }
```

Output Compile Error

Exception #1

This behavior is scoped to either the root or to a specific call to a module, meaning you can re-define a variable within a module without affecting its value outside of it. However, all instances within that call will behave as described above with the last-set value being used throughout.

Example:

```
p = 4;
test(5);
echo(p);
/*
 * we start with p = 4.  We step to the next command 'test(5)', which calls the 'test' module.
 * The 'test' module calculates two values for 'p', but the program will ONLY display the final value.
 * There will be two executions of echo(p) inside 'test' module, but BOTH will display '9' because it is the FINAL
 * calculated value inside the module. ECHO: 9   ECHO: 9
 *
 * Even though the 'test' module calculated value changes for 'p', those values remained inside the module.
 * Those values did not continue outside the 'test' module.  The program has now finished 'test(5)' and moves to the next command 'echo(p)'.
 * The call 'echo(p)' would normally display the original value of 'p'=4.
 * Remember that the program will only show the FINAL values.  It is the next set of commands that produce the final values....which is ECHO: 6
 */
p = 6;
test(8);
echo(p);
/*
 * We now see 'p=6', which is a change from earlier.  We step to the next command 'test(8)', which calls the 'test' module.
 * Again, the 'test' module calculates two values for 'p', but the program will ONLY display the final value.
 * There will be two executions of echo(p) inside 'test' module, but BOTH will display '12' because it is the FINAL
 * compiled value that was calculated inside the module.
 * Therefore, both echo(p) statements will show the final value of '12' ;
 * Remember that the 'test' module final values for 'p' will remain inside the module.  They do not continue outside the 'test' module.
 * ECHO:12   ECHO:  12
 *
 * The program has now finished 'test(8)' and moves to the next command 'echo(p)'.
 * Remember at compile that the pgm will show the FINAL values.  The first value of 'echo(p)' would have showed a value of '4'...
 * However, at compile time the final value of 'echo(p)' was actually '6'.  Therefore, '6' will be shown on both echo(p) statements.
 * ECHO 6
 */

module test(q)
{
    p = 2 + q;
    echo(p);

    p = 4 + q;
    echo(p);
}
```

Output

```
ECHO: 9
ECHO: 9
```

```
ECHO: 6
ECHO: 12
ECHO: 12
ECHO: 6
```

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.

Exception #2

See the assign, which provides for a more tightly scoped changing of values.

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 dxf_cross will read the intersection of two lines on a layer you specify and return the intersection point. This means that the point must be given with two lines in the DXF file, and not a point entity.

```
OriginPoint = dxf_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 (http://www.openscad.org/).

For Loop

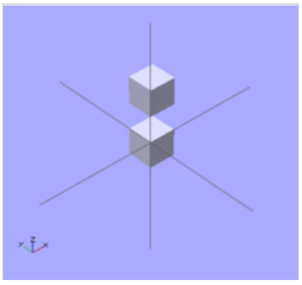
Iterate over the values in a vector or range.

Vector version: for (variable=<vector>) <do_something> - <variable> is assigned to each successive value in the vector
Range version: for (variable=<range>) <do_something>

Nested loops : for (variable1 = <range or vector>, variable2 = <range or vector>) <do something, using both variables>
for loops can be nested, just as in normal programs. A shorthand is that both iterations can be given in the same for statement

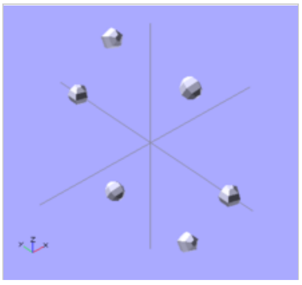
Usage example 1 - iteration over a vector:

```
for (z = [-1, 1]) // two iterations, z = -1, z = 1
{
    translate([0, 0, z])
    cube(size = 1, center = false);
}
```


OpenSCAD iteration over a vector

Usage example 2a - iteration over a range:

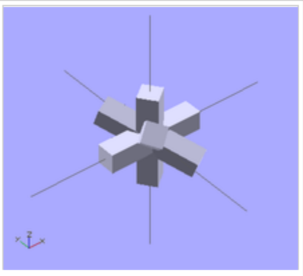
```
for ( i = [0 : 5] )
{
    rotate( i * 360 / 6, [1, 0, 0])
    translate([0, 10, 0])
    sphere(r = 1);
}
```


OpenSCAD iteration over a range)

Usage example 2b - iteration over a range specifying an increment:

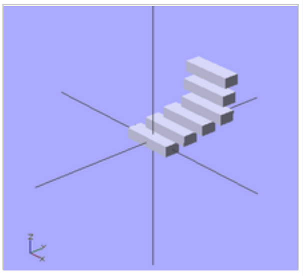
```
// Note: The middle parameter in the range designation
// ('0.2' in this case) is the 'increment-by' value
// Warning: Depending on the 'increment-by' value, the
// real end value may be smaller than the given one.
for ( i = [0 : 0.2 : 5] )
{
    rotate( i * 360 / 6, [1, 0, 0])
    translate([0, 10, 0])
    sphere(r = 1);
}
```

Usage example 3 - iteration over a vector of vectors (rotation):


OpenSCAD for loop (rotation)

```
for(i = [ [  0,  0,   0],
          [ 10, 20, 300],
          [200, 40,  57],
          [ 20, 88,  57] ])
{
    rotate(i)
    cube([100, 20, 20], center = true);
}
```

Usage example 4 - iteration over a vector of vectors (translation):


OpenSCAD for loop (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);
}
```

Nested loop example

```
 for (xpos=[0:3], ypos = [2,4,6]) // do twelve iterations, using each xpos with each ypos
  translate([xpos*ypos, ypos, 0]) cube([0.5, 0.5, 0.5]);
```

Intersection For Loop

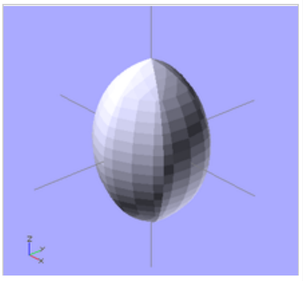
Iterate over the values in a vector or range and take an intersection of the contents.

Note: `intersection_for()` is a work around because of an issue that you cannot get the expected results using a combination of the standard `for()` and `intersection()` statements. The reason is that `for()` does an implicit `union()` of the contents.

Parameters

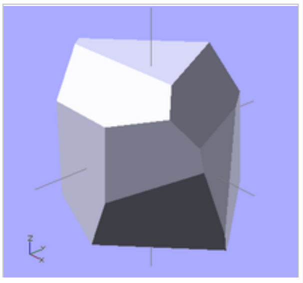
<loop variable name>
    Name of the variable to use within the **for** loop.

Usage example 1 - loop over a range:


OpenSCAD Intersection for

```
intersection_for(n = [1 : 6])
{
    rotate([0, 0, n * 60])
    {
        translate([5,0,0])
        sphere(r=12);
    }
}
```

Usage example 2 - rotation :


OpenSCAD Intersection for (rotation)

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

If Statement

Conditionally evaluate a sub-tree.

Parameters

- The boolean expression that should be used as condition

NOTE:

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

```
if (a=b) dosomething();  // WRONG - this will FAIL to be processed without any error message
if (a==b) dosomething(); // CORRECT - this will do something if a equals b
```

NOTE:

Assignment is not allowed within either branch of an if statement. Consider using the ternary operator 'condition ? consequent: alternative'.

```
// WRONG - this will FAIL to be processed with a syntax error message
if (condition)
{
    x = consequent;
} else {
    x = alternative;
}
```

```
// CORRECT - this will set 'x' to either 'consequent' or 'alternative'
x = condition ? consequent : alternative;
```

Usage example:

```
if (x > y)
{
    cube(size = 1, center = false);
} else {
    cube(size = 2, center = true);
}
```

Assign Statement

Set variables to a new value for a sub-tree.

Parameters

- The variables that should be (re-)assigned

Usage example:

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

Update: [***Note:*** *Requires version **2015.03***]

Starting with this version, assign() has been deprecated as it is no longer needed; variables can be assigned anywhere.

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.

Relational Operators

All relational operator take numbers as operands and produce a Boolean value. The equal and not-equal operators can also compare Boolean values.

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

Logical Operators

All logical operators take Boolean values as operands and produce a Boolean value.

&&	Logical AND
||	Logical OR
!	Logical NOT

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.

? :	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.

Vector-Number Operators

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

*	multiply all vector elements by number
/	divide all vector elements by number

Vector Operators

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

+	add element-wise
-	subtract element-wise

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

Vector Dot-Product Operator

The vector dot-product operator takes two vectors as operands and produces a scalar.

*	sum of vector element products

Matrix Multiplication

Multiplying a matrix by a vector, vector by matrix and matrix by matrix

*	matrix/vector multiplication

Trigonometric Functions

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

Contents

(http://pubs.opengroup.org/onlinepubs/009695399/basedefs/math.h.html) & acos (http://pubs.opengroup.org/onlinepubs/009695399/functions/acos.html)

cos

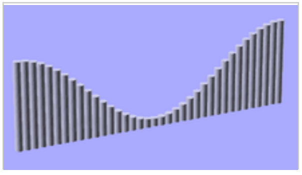
Mathematical **cosine** function of degrees. See Cosine

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);
```


OpenSCAD Cos Function

sin

Mathematical **sine** function. See Sine

Parameters

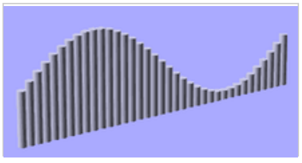
<degrees>
: Decimal. Angle in degrees.

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

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);
}
```

acos

Mathematical **arccosine**, or **inverse cosine**, expressed in degrees. See: Inverse trigonometric functions

asin

Mathematical **arcsine**, or **inverse sine**, expressed in degrees. See: Inverse trigonometric functions

atan

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

atan2

Mathematical **two-argument atan** function, taking y as its first argument. Returns the principal value of the arc tangent of y/x, expressed in degrees. See: atan2

Other Mathematical Functions

abs

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

Usage examples:

```
abs(-5.0);
abs(0);
abs(8.0);
```

Results:

```
5.0
0.0
8.0
```

ceil

Mathematical **ceiling** function. ceil(x) is the smallest integer not less than x.

See: Ceil Function

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

concat

[***Note:*** *Requires version **2015.03***]

Return a vector containing the arguments.

Where an argument is a vector the elements of the vector are individually added 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"
```

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) will produce 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
```

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
```

floor

Mathematical **floor** function. floor(x) = is the largest integer not greater than x

See: Floor Function

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

ln

Mathematical **natural logarithm**. See: Natural logarithm

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 doIt(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);
        }
 }

doIt(5);        // equivalent to [5,5,5]
doIt([5,5,5]);  // similar to cube(5) v's cube([5,5,5])
```

let

[***Note:*** *Requires version* ***2015.03***]

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]
```

log

Mathematical **logarithm**. See: Logarithm

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

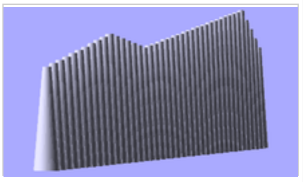
Notes
There is a bug where out-of-range keys will return the first value in the list. Newer versions of Openscad should use the top or bottom end of the table as appropriate instead.

Usage example:

- Will create a sort of 3D chart made out of cylinders of different height.

```
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

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}...)
max(vector)
```

<n>
: Two or more decimals

<vector>
: Single vector of decimals (requires OpenSCAD version 2014.06 or later).

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
```

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}...)
min(vector)
```

<n>
: Two or more decimals

<vector>
: Single vector of decimals (requires OpenSCAD version 2014.06 or later).

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
```

Looking for **mod** - it's not a function, see modulo operator (%)

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
```

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 equals calculating the cube root of 125
// result: ECHO: 5
```

rands

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.

Usage Examples:

```
// get a single number
single_rand = rands(0,10,1)[0];
echo(single_rand);
```

```
// get a vector of 4 numbers
seed=42;
random_vect=rands(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]
```

round

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

Some examples:

```
round(x.5) = x+1.
round(x.49) = x.
round(-(x.5)) = -(x+1).
round(-(x.49)) = -x.

round(5.4); //-> 5
round(5.5); //-> 6
round(5.6); //-> 6
```

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
```

sqrt

Mathematical **square root** function.

Usage Examples:

```
translate([sqrt(100),0,0])sphere(100);
```

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."
```

chr

[***Note:*** *Requires version* ***2015.03***]

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 will 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.

Also See search()

search() for text searching.

cube

Creates a cube at the origin of the coordinate system. When center is true the cube will be centered on the origin, otherwise it is created in the first octant. The argument names are optional if the arguments are given in the same order as specified in the parameters

Parameters

size

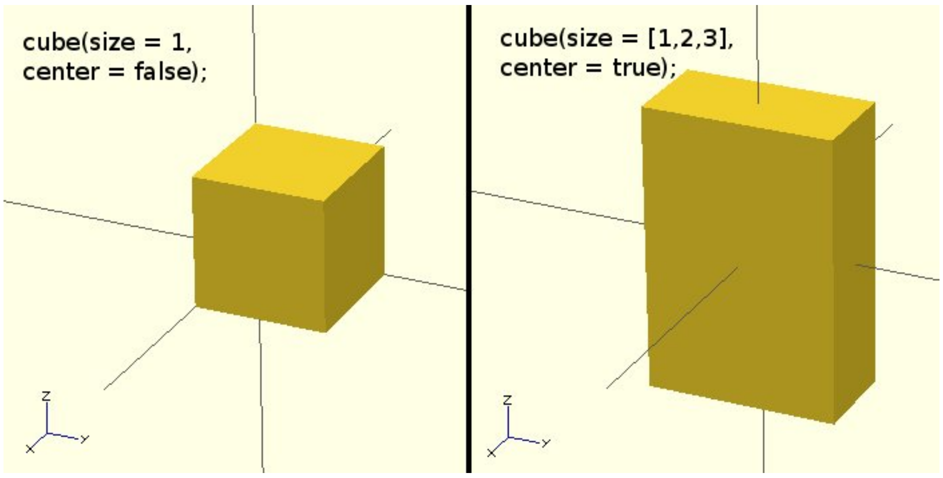
Decimal or 3 value array. If a single number is given, the result will be a cube with sides of that length. If a 3 value array is given, then the values will correspond to the lengths of the X, Y, and Z sides. Default value is 1.

center

Boolean. This determines the positioning of the object. If true, object is centered at (0,0,0). Otherwise, the cube is placed in the positive quadrant with one corner at (0,0,0). Defaults to false

Usage examples:

```
cube(size = 1, center = false);
cube(size = [1,2,3], center = true);
```



sphere

Creates a sphere at the origin of the coordinate system. The argument name is optional.

Parameters

r

Radius. This is the radius of the sphere. The resolution of the sphere will be 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.

(NOTE: d is only available in versions later than 2014.03. Debian is currently known to be behind this)

$fa

Fragment angle in degrees

$fs

Fragment size in mm

$fn

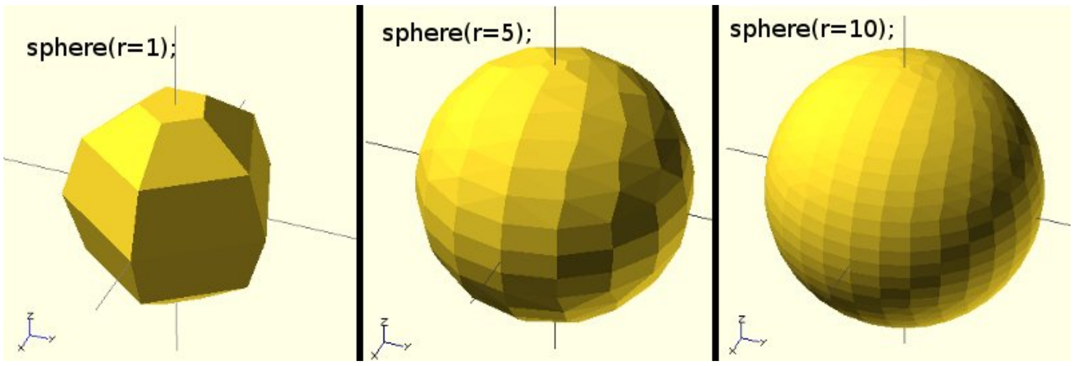
Resolution

Usage Examples

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

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

```
// will also create 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);
```



cylinder

Creates a cylinder or cone at the origin of the coordinate system. A single radius (r) makes a cylinder, two different radi (r1, r2) make a cone.

Parameters

h
: Decimal. This is the height of the cylinder. Default value is 1.

r
: Decimal. The radius of both top and bottom ends of the cylinder. Use this parameter if you want plain cylinder. Default value is 1.

r1
: Decimal. This is the radius of the cone on bottom end. Default value is 1.

r2
: Decimal. This is the radius of the cone on top end. Default value is 1.

d
: Decimal. The diameter of both top and bottom ends of the cylinder. Use this parameter if you want plain cylinder. Default value is 1.

d1
: Decimal. This is the diameter of the cone on bottom end. Default value is 1.

d2
: Decimal. This is the diameter of the cone on top end. Default value is 1.

center
: boolean. If true will center the height of the cone/cylinder around the origin. Default is false, placing the base of the cylinder or r1 radius of cone at the origin.

$fa
: The angle (in degrees) from one fragment to the next. See OpenSCAD_User_Manual/Other_Language_Features.

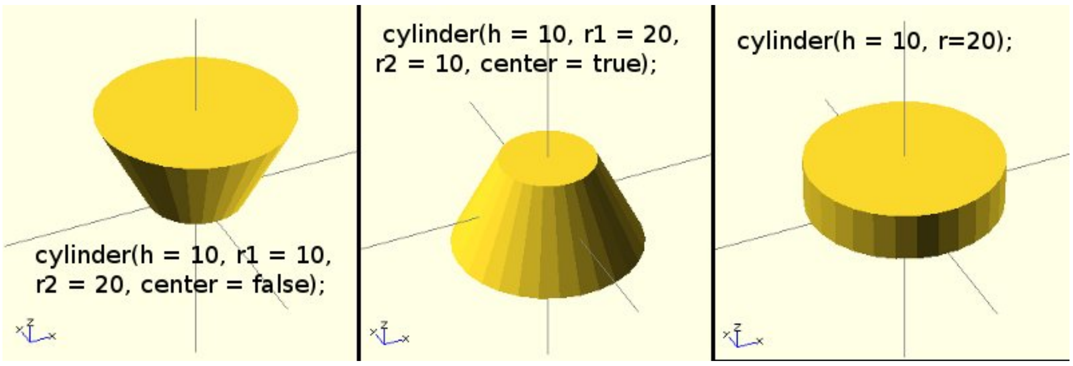
$fs
: The circumferential length of each fragment. See OpenSCAD_User_Manual/Other_Language_Features.

$fn
: The fixed number of fragments to use. See OpenSCAD_User_Manual/Other_Language_Features.

(NOTE: d,d1,d2 are only available in version later than 2014.03. Debian is currently know to be behind this)

Usage Examples

```
cylinder(h = 10, r=20);
cylinder(h = 10, r=20, $fs=6);
cylinder(h = 10, r1 = 10, r2 = 20, center = false);
cylinder(h = 10, r1 = 20, r2 = 10, center = true);
cylinder(h = 10, d=40);
cylinder(h = 10, d=40, $fs=6);
cylinder(h = 10, d1 = 20, d2 = 40, center = false);
cylinder(h = 10, d1 = 40, d2 = 20, center = true);
```


cylinder(h = 10, r1 = 10, r2 = 20, center = false);
cylinder(h = 10, r1 = 20, r2 = 10, center = true);
cylinder(h = 10, r=20);

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 behaviour is illustrated in a diagram (https://camo.githubusercontent.com/533961dfae3fd5643f3474345e4179a8a328dcf9/68747470733a2f2f662e636c6f75642e6769746875622e636f6d2f6173736574732f3 . Discussion regarding optionally changing this behaviour happening in a Pull Request (https://github.com/openscad/openscad/pull/599)

polyhedron

Create a polyhedron with a list of points and a list of faces. The point list is all the vertices of the shape, the faces list is how the points relate to the surfaces of the polyhedron.

note: if your version of OpenSCAD is lower than 2014.03 replace "faces" with "triangles" in the below examples

Parameters

points
: vector of points or vertices (each a 3 vector).

triangles
: (*deprecated in version 2014.03, use faces*) vector of point triplets (each a 3 number vector). Each number is the 0-indexed point number from the point vector.

faces
: (*introduced in version 2014.03*) vector of point n-tuples with n >= 3. Each number is the 0-indexed point number from the point vector. That is, faces=[[0,1,4]] specifies a triangle made from the first, second, and fifth point listed in points. When referencing more than 3 points in a single tuple, the points must all be on the same plane.

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.

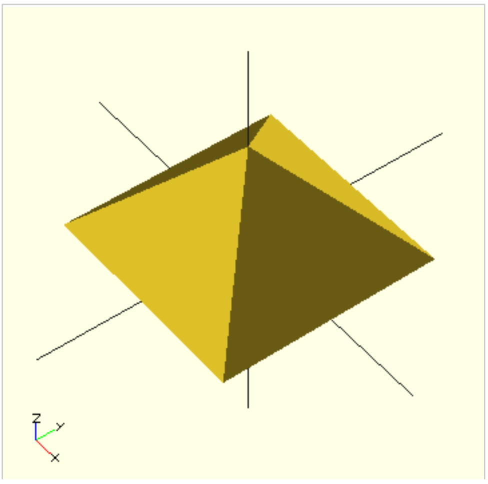
Syntax example

```
polyhedron(points = [ [x, y, z], ... ], faces = [ [p1, p2, p3..], ... ], convexity = N);
```

Point ordering for faces When looking at the face from the outside inwards, the points must be clockwise. You can rearrange the order of the points or the order they are referenced in each tuple. The order of faces is immaterial. Note that if your polygons are not all oriented the same way OpenSCAD will either print an error or crash completely, so pay attention to the vertex ordering. Again, remember that the 'pN' components of the faces vector are 0-indexed references to the elements of the points vector.

Example, a 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
 );
```



A simple polyhedron, square based pyramid

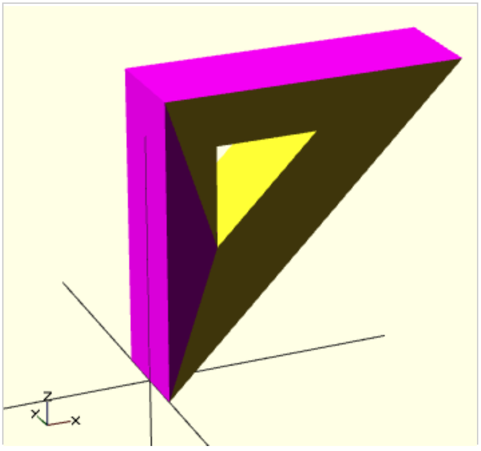
polyhedron - polygon orientation issues

An example of a more complex polyhedron, and showing how to fix polyhedrons with badly oriented polygons.

When you select 'Thrown together' from the view menu and **compile** the design (**not** compile and render!) you will see a preview with 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]
                 ]
     );
```

Beginner's tip:

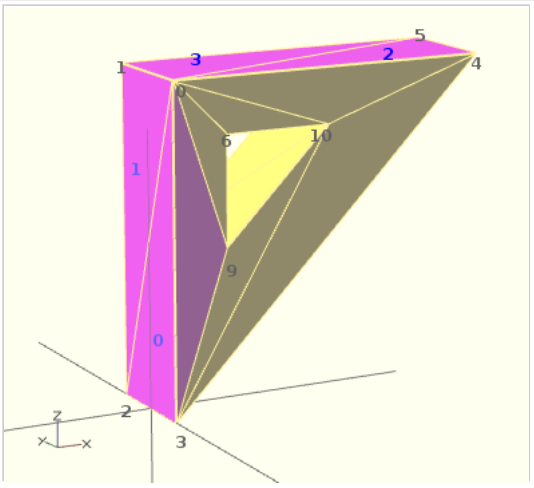
If you don't really understand "orientation", try to identify the mis-oriented pink faces and then permute the references to the points vectors until you get it right. E.g. in the above example, the third triangle (*[0,4,5]*) was wrong and we fixed it as *[4,0,5]*. In addition, you may select "Show Edges" 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 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 hold the face and the fingers of your hand curls is the same order as the points, then your thumb points outwards.



Polyhedron with badly oriented polygons

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 given an identifier starting at 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 identifier) 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. 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 '{' & '}' to enclose the subtree e.g.

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

Advanced concept

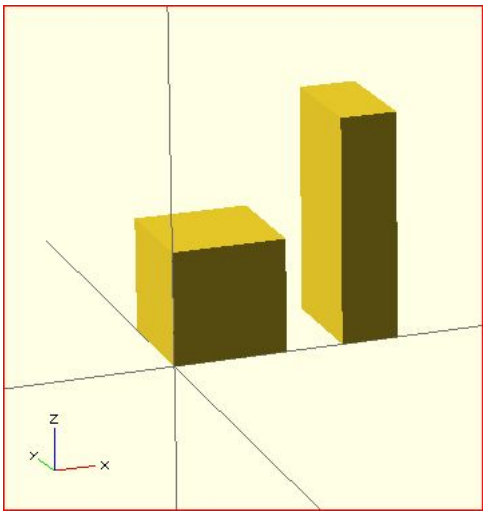
As OpenSCAD uses different libraries to implement capabilities this can introduce some inconsistencies to the F5 preview behaviour of transformations. Traditional transforms (translate, rotate, scale, mirror & multimatrix) are performed using OpenGL in preview, while other more advanced transforms, such 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.

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);
```



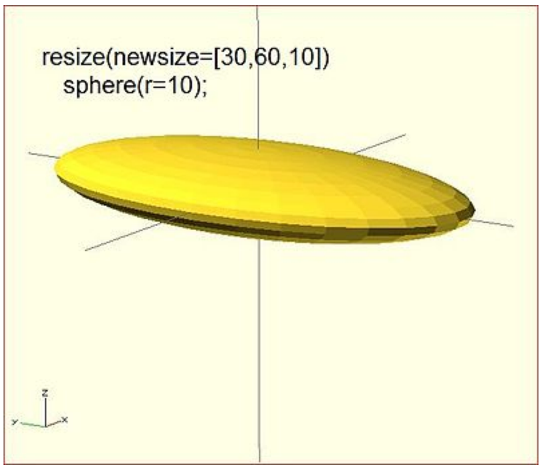
resize

resize() is available since OpenSCAD 2013.06. It modifies the size of the child object to match the given x,y, and z.

There is a bug with shrinking in the 2013.06 release, that will be fixed in the next release.

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 will auto-scale 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]);
```

rotate

Rotates its child 'a' degrees about the origin 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.

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

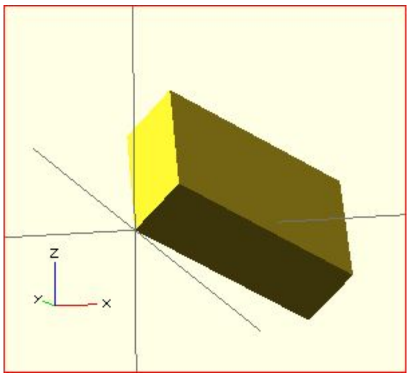
```
rotate([0,180,0]) { ... }
```

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]) { ... }
```

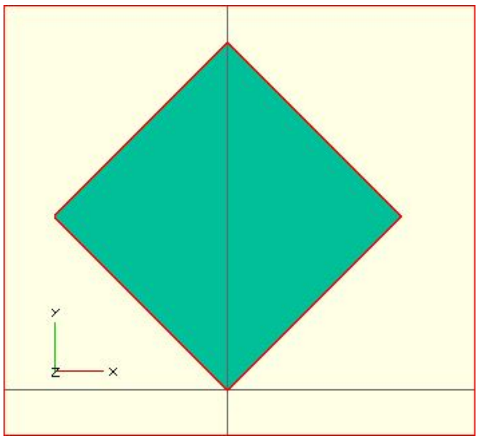
When specifying mutiple axis, 'v'is a vector defining an arbitrary axis for rotation; this is different from the *multiple axis* above. For example, rotate your object 45 degrees around the axis defined by the vector [1,1,0],

```
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);
```



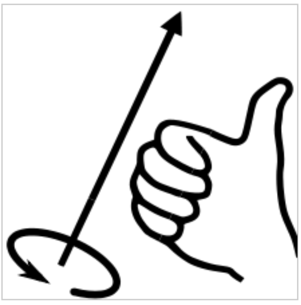
Rotation rule help

For the case of:

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

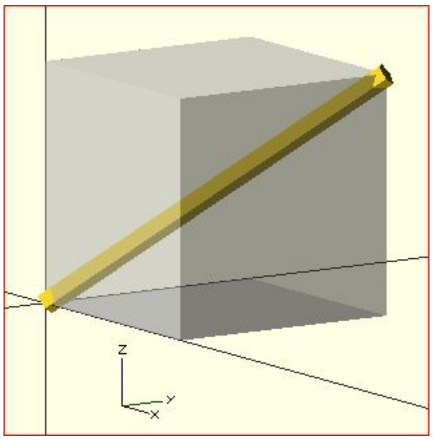
"a" is a rotation about the X axis, from the +Z axis, toward the -Y axis. NOTE: NEGATIVE Y.
"b" is a rotation about the Y axis, from the +Z axis, toward the +X axis.
"c" is a rotation about the Z axis, from the +X axis, toward the +Y axis.

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


Right-hand grip rule

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):

```
%cube(10);
x= 10; y = 10; z = 10; // point coordinates of end of cylinder
//
length = sqrt( pow(x, 2) + pow(y, 2) + pow(z, 2) );
b = acos(z/length);
c = (x==0) ? sign(y)*90 : ( (x>0) ? atan(y/x) : atan(y/x)+180 );
//
rotate([0, b, c])
    cylinder(h=length, r=0.5);
```



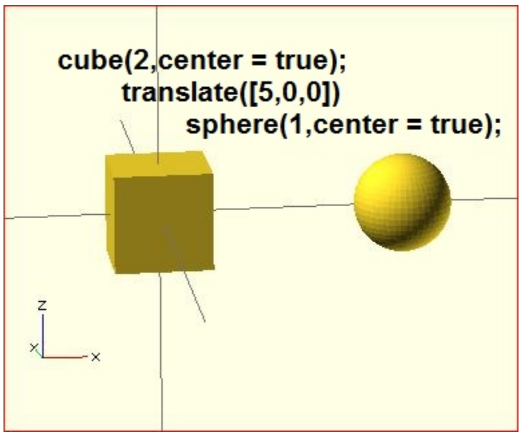
translate

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

IExample

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

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



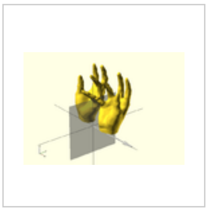
mirror

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:

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

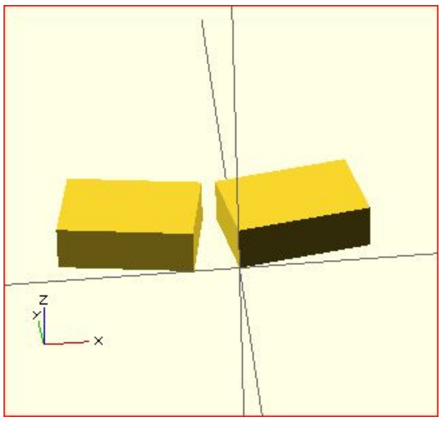
Examples


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

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

`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]);
```



multmatrix

Multiplies the geometry of all child elements with the given 4x4 transformation matrix.

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

Example (translates by [10, 20, 30]):

```
multmatrix(m = [ [1, 0, 0, 10],
                 [0, 1, 0, 20],
                 [0, 0, 1, 30],
                 [0, 0, 0,  1]
               ]) cylinder(r=10.0,h=10);
```

Example (rotates by 45 degrees in XY plane and translates by [10,20,30]):

```
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);
}
```

More?

Learn more about it here:

- Affine Transformations (http://en.wikipedia.org/wiki/Transformation_matrix#Affine_transformations) on wikipedia
- http://www.senocular.com/flash/tutorials/transformmatrix/

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 will default to 1.0 (opaque) if not specified.

Function signature:

```
color( [r, g, b, a] ) { ... }
color( [r, g, b], a=1.0 ) { ... } // since v. 2011.12 (?)
color( colorname, 1 ) { ... } // since v. 2011.12 ( fails in 2014.03; use color( "colorname", #) were # is the alpha )
```

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 ]) { ... }`

Since version *2011.12*, colors can also be defined by name (case **in**sensitive). 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);`

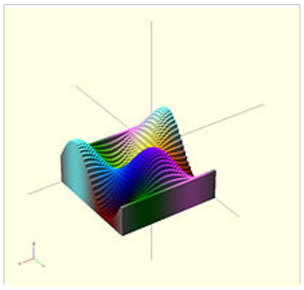
The available color names are taken from the World Wide Web consortium's SVG color list (http://www.w3.org/TR/css3-color/). A chart of the color names is as follows,
(note that both spellings of grey/gray including slategrey/slategray etc are valid):

Purples	**Blues**	**Greens**	**Yellows**	**Whites**
Lavender	Aqua	GreenYellow	Gold	White
Thistle	Cyan	Chartreuse	Yellow	Snow
Plum	LightCyan	LawnGreen	LightYellow	Honeydew
Violet	PaleTurquoise	Lime	LemonChiffon	MintCream
Orchid	Aquamarine	LimeGreen	LightGoldenrodYellow	Azure
Fuchsia	Turquoise	PaleGreen	PapayaWhip	AliceBlue
Magenta	MediumTurquoise	LightGreen	Moccasin	GhostWhite
MediumOrchid	DarkTurquoise	MediumSpringGreen	PeachPuff	WhiteSmoke
MediumPurple	CadetBlue	SpringGreen	PaleGoldenrod	Seashell
BlueViolet	SteelBlue	MediumSeaGreen	Khaki	Beige
DarkViolet	LightSteelBlue	SeaGreen	DarkKhaki	OldLace
DarkOrchid	PowderBlue	ForestGreen	**Browns**	FloralWhite
DarkMagenta	LightBlue	Green	Cornsilk	Ivory
Purple	SkyBlue	DarkGreen	BlanchedAlmond	AntiqueWhite
Indigo	LightSkyBlue	YellowGreen	Bisque	Linen
DarkSlateBlue	DeepSkyBlue	OliveDrab	NavajoWhite	LavenderBlush
SlateBlue	DodgerBlue	Olive	Wheat	MistyRose
MediumSlateBlue	CornflowerBlue	DarkOliveGreen	BurlyWood	**Grays**
Pinks	RoyalBlue	MediumAquamarine	Tan	Gainsboro
Pink	Blue	DarkSeaGreen	RosyBrown	LightGrey
LightPink	MediumBlue	LightSeaGreen	SandyBrown	Silver
HotPink	DarkBlue	DarkCyan	Goldenrod	DarkGray
DeepPink	Navy	Teal	DarkGoldenrod	Gray
MediumVioletRed	MidnightBlue	**Oranges**	Peru	DimGray
PaleVioletRed	**Reds**	LightSalmon	Chocolate	LightSlateGray
	IndianRed	Coral	SaddleBrown	SlateGray
	LightCoral	Tomato	Sienna	DarkSlateGray
	Salmon	OrangeRed	Brown	Black
	DarkSalmon	DarkOrange	Maroon	
	LightSalmon	Orange		
	Red			
	Crimson			
	FireBrick			
	DarkRed			

Example

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)] );
  }
}
```


A 3-D multicolor sine wave

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

Chart based on "Web Colors" from Wikipedia (http://en.wikipedia.org/wiki/Web_colors)

offset

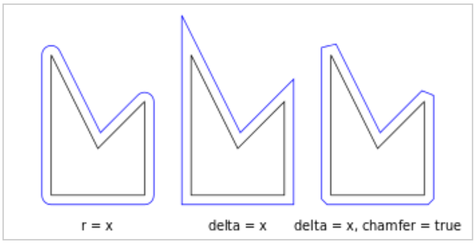
[***Note:** Requires version **2015.03***]

Offset allows moving polygon outlines outward or inward by a given amount. Only on 2D Primitives.

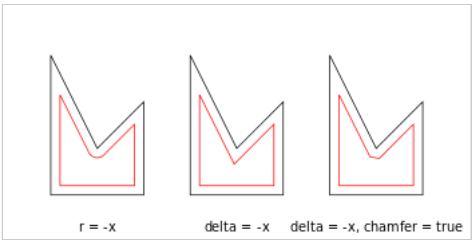
Parameters

r | delta
Double. Amount to offset the polygon. When negative, the polygon is offset inwards. The parameter r specifies the radius that is used to generate rounded corners, using delta gives straight edges.

chamfer
Boolean. (default *false*) When using the delta parameter, this flag defines if edges should be chamfered or not.



Positive r/delta value

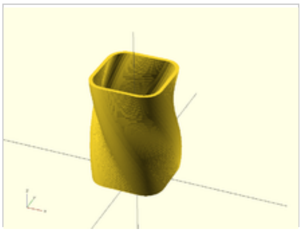


Negative r/delta value

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

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 1: Result.

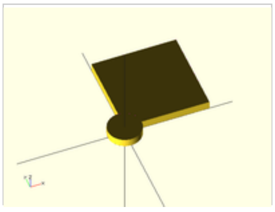
minkowski

Displays the minkowski sum (http://www.cgal.org/Manual/latest/doc_html/cgal_manual/Minkowski_sum_3/Chapter_main.html) of child nodes.

Usage example:

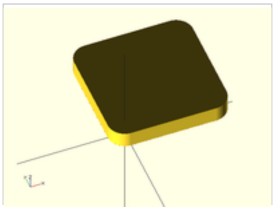
Say you have a flat box, and you want a rounded edge. There are many ways to do this, but minkowski is very elegant. Take your box, and a cylinder:

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


A box and a cylinder

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);
}
```

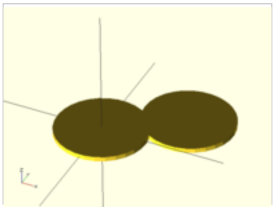

Minkowski sum of the box and cylinder

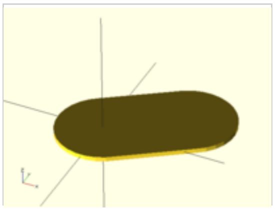
hull

Displays the convex hull (http://www.cgal.org/Manual/latest/doc_html/cgal_manual/Convex_hull_2/Chapter_main.html) of child nodes.

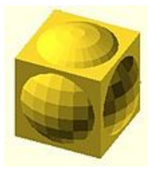
Usage example:

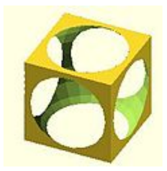
```
hull() {
   translate([15,10,0]) circle(10);
   circle(10);
 }
```

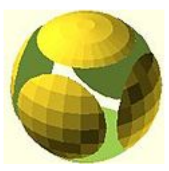

Two cylinders

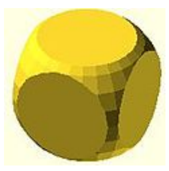

Convex hull of two cylinders

boolean overview


union sphere + cube


difference cube - sphere


difference sphere - cube


intersection sphere - (sphere - cube)

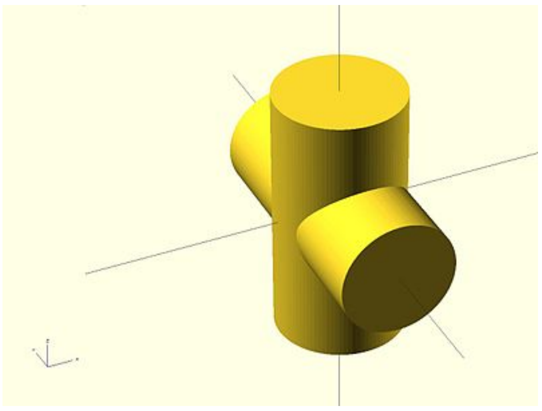
```
examples:
union(){  // cube and sphere
  cube(12, center=true);
  sphere(8);
}
difference(){   // cube and not sphere
  cube(12, center=true);
  sphere(8);
}
difference(){  // sphere and not cube
  sphere(8);
  cube(12, center=true);
}
intersection(){  // cube and sphere
  cube(12, center=true);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

union

Creates a union of all its child nodes. This is the **sum** of all children (logical **or**).

```
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);
}
```

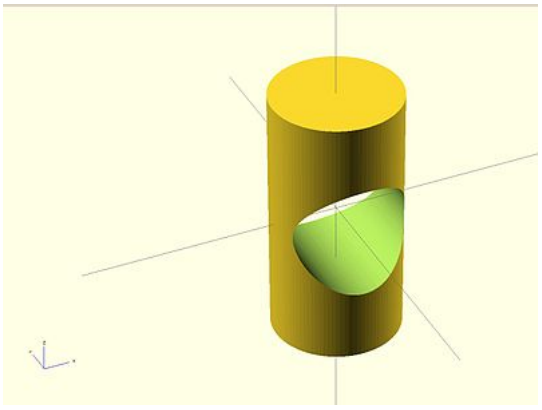
Remark: union is implicit when not used. But it is mandatory, for example, in difference to group first child nodes into one.



difference

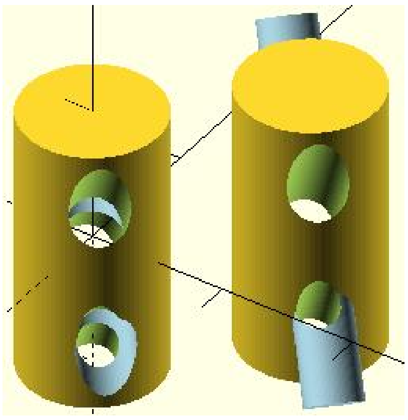
Subtracts the 2nd (and all further) child nodes from the first one (logical **and not**).

```
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);
}
```



Further examples with multiple children.
Note the effect of adding a union in the second instance.

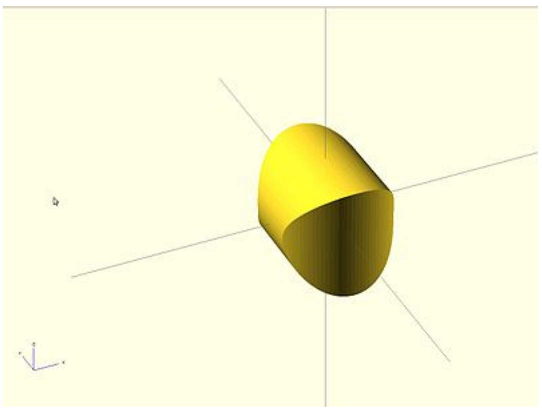
```
Usage example:
 $fn=90;
 difference(){
     cylinder(r=5,h=20,center=true);
   rotate([00,140,-45]) color("LightBlue")
     cylinder(r=2,h=25,center=true);
   rotate([00,40,-50])
     cylinder(r=2,h=30,center=true);
   translate([0,0,-10])rotate([00,40,-50])
     cylinder(r=1.4,h=30,center=true);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

 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);
   }
 }
```



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.

```
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);
}
```

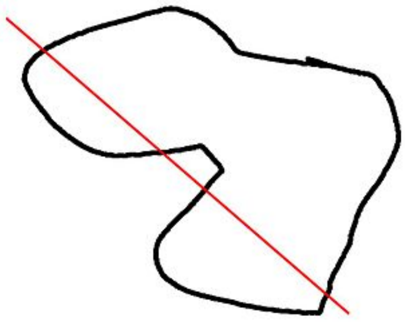


render

Always calculate the CSG model for this tree (even in OpenCSG preview mode).

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

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.

Modifier characters are used to change the appearance or behaviours of child nodes. They are particularly useful in debugging where they can be used to highlight specific objects, or include or exclude them from rendering.

Advanced concept

As OpenSCAD uses different libraries to implement capabilities this can introduce some inconsistencies to the F5 preview behaviour 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.

Note: The color changes triggered by character modifiers will only be shown in "Compile" mode not "Compile and Render (CGAL)" mode. (As per the color section.)

Background Modifier

Ignore this subtree for the normal rendering process and draw it in transparent gray (all transformations are still applied to the nodes in this tree).

Because the marked subtree is completely ignored, it might have unexpected effects in case it's used for example with the first object in a difference(). In that case this object will be rendered in transparent gray, but it will *not* be the base for the difference()!

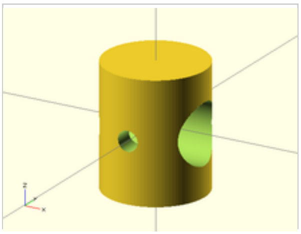
Usage

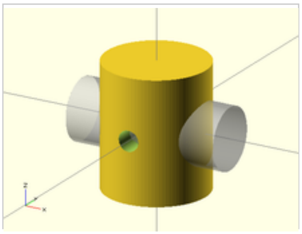
```
% { ... }
```

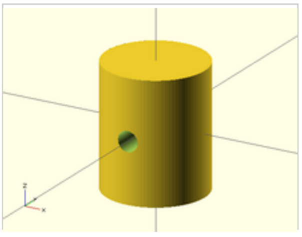
Example

```
difference() {
        cylinder (h = 12, r=5, center = true, $fn=100);
        // first object that will subtracted
        rotate ([90,0,0]) cylinder (h = 15, r=1, center = true, $fn=100);
        // second object that will be subtracted
        %rotate ([0,90,0]) cylinder (h = 15, r=3, center = true, $fn=100);
}
```

Example Output


Output without the modifer.


Output with modifier added.


Rendered Model.

Debug Modifier

Use this subtree as usual in the rendering process but also draw it unmodified in transparent pink.

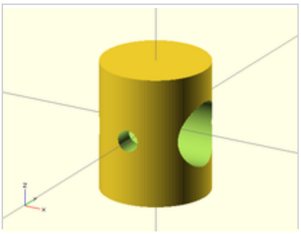
Usage

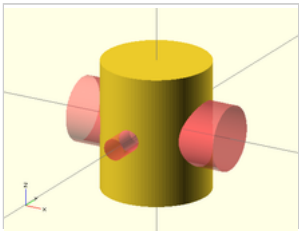
```
# { ... }
```

Example

```
difference() {
        // start objects
        cylinder (h = 12, r=5, center = true, $fn=100);
        // first object that will subtracted
        #rotate ([90,0,0]) cylinder (h = 15, r=1, center = true, $fn=100);
        // second object that will be subtracted
        #rotate ([0,90,0]) cylinder (h = 15, r=3, center = true, $fn=100);
}
```

Example Output


Output without the modifer.


Output with modifier added.

Root Modifier

Ignore the rest of the design and use this subtree as design root.

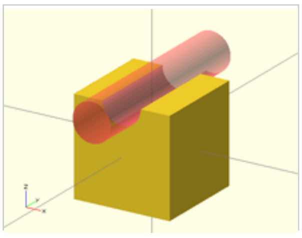
Usage

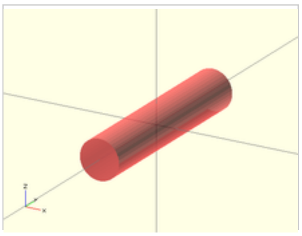
```
! { ... }
```

Example

```
difference() {
        cube(10, center = true);
        translate([0, 0, 5]) {
                !rotate([90, 0, 0]) {
                        #cylinder(r = 2, h = 20, center = true, $fn = 40);
                }
        }
}
```

Example Output


Output without the modifer.


Output with modifier added.

As shown in the example output with the root modifier active, the rotate() is executed as it's part of the subtree marked with the root modifier, but the translate() has no effect.

Disable Modifier

Simply ignore this entire subtree.

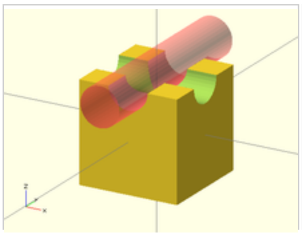
Usage

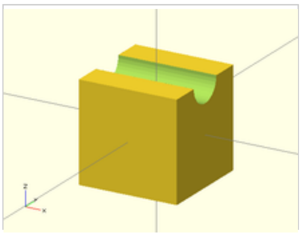
```
* { ... }
```

Example

```
difference() {
        cube(10, center = true);
        translate([0, 0, 5]) {
                rotate([0, 90, 0]) {
                        cylinder(r = 2, h = 20, center = true, $fn = 40);
                }
                *rotate([90, 0, 0]) {
                        #cylinder(r = 2, h = 20, center = true, $fn = 40);
                }
        }
}
```

Example Output


Output without the modifer.


Output with modifier added.

The disable modifier allows to comment out one or multiple subtrees. Compared to using the usual line or multi-line comments, it's aware of the hierarchical structure which makes it easier to disable even larger trees without the need to search for the end of the subtree.

usage

Defining your own module (roughly comparable to a macro or a function in other languages) is a powerful way to reuse procedures.

```
module hole(distance, rot, size) {
    rotate(a = rot, v = [1, 0, 0]) {
        translate([0, distance, 0]) {
            cylinder(r = size, h = 100, center = true);
        }
    }
}
```

In this example, passing in the parameters distance, rot, and size allow you to reuse this functionality multiple times, saving many lines of code and rendering your program much easier to read.

You can instantiate the module by passing values (or formulas) for the parameters just like a C function call:

```
hole(0, 90, 10);
```

children

The child nodes of the module instantiation can be accessed using the children() statement within the module. The number of module children can be accessed using the $children variable.

Parameters

empty
: select all the children

index
: integer. select one child, at index value. Index start at 0 and should be less than or equal to $children-1.

vector
: vector of integer. select children with index in vector. Index should be between 0 and $children-1.

range
: [<start>:<end>] or [<start>:<increment>:<end>]. select children between <start> to <end>, incremented by <increment> (default 1).

Deprecated child() module

Up to release 2013.06 the now deprecated `child()` module was used instead. This can be translated to the new children() according to the table:

up to 2013.06	2014.03 and later
child()	children(0)
child(x)	children(x)
for (a = [0:$children-1]) child(a)	children([0:$children-1])

Examples

Transfer all children to another module:

```
// rotate to other center point:
module rz(angle, center=undef) {
  translate(center)
     rotate(angle)
        translate(-center)
           children();
}

rz(15, [10,0]) sphere(30);
```

Use the first child, multiple time:

```
module lineup(num, space) {
  for (i = [0 : num-1])
    translate([ space*i, 0, 0 ]) children(0);
}

lineup(5, 65) sphere(30);
```

If you need to make your module iterate over all children you will need to make use of the $children variable, e.g.:

```
module elongate() {
  for (i = [0 : $children-1])
    scale([10 , 1, 1 ]) children(i);
}

elongate() { sphere(30); cube([10,10,10]); cylinder(r=10,h=50); }
```

arguments

One can specify default values for the arguments:

```
module house(roof="flat",paint=[1,0,0]){
  color(paint)
  if(roof=="flat"){
    translate([0,-1,0])
    cube();
  } else if(roof=="pitched"){
    rotate([90,0,0])
    linear_extrude(height=1)
    polygon(points=[[0,0],[0,1],[0.5,1.5],[1,1],[1,0]],paths=[ [0,1,2,3,4] ]);
  } else if(roof=="domical"){
    translate([0,-1,0])
    union(){
      translate([0.5,0.5,1]) sphere(r=0.5,$fn=20);
      cube();
    }
  }
}
```

And then use one of the following ways to supply the arguments

```
union(){
  house();
  translate([2,0,0]) house("pitched");
  translate([4,0,0]) house("domical",[0,1,0]);
  translate([6,0,0]) house(roof="pitched",paint=[0,0,1]);
  translate([8,0,0]) house(paint=[0,0,0],roof="pitched");
  translate([10,0,0]) house(roof="domical");
  translate([12,0,0]) house(paint=[0,0.5,0.5]);
}
```

For including code from external files in OpenSCAD, there are two commands available:

- `include <filename>` acts as if the contents of the included file were written in the including file, and
- `use <filename>` imports modules and functions, but does not execute any commands other than those definitions.

Library files are searched for in the same folder as the design was open from, or in the library folder of the OpenSCAD installation. You can use a relative path specification to either. If they lie elsewhere you must give the complete path. Newer versions have predefined user libraries, see the OpenSCAD_User_Manual/Libraries page, which also documents a number of library files included in OpenSCAD.

Windows and Linux/Mac use different separators for directories. Windows uses \, e.g. directory\file.ext, while the others use /, e.g. directory/file.ext. This could lead to cross platform issues. However OpenSCAD on Windows correctly handles the use of /, so using / in all **include** or **use** statements will work on all platforms.

Using **`include <filename>`** allows default variables to be specified in the library. These defaults can be overridden in the main code. An openscad variable only has one value during the life of the program. When there are multiple assignments it takes the last value, but assigns when the variable is first created. This has an effect when assigning in a library, as any *variables* which you later use to change the default, must be assigned before the include statement. See the second example below.

A library file for generating rings might look like this (defining a function and providing an example):

ring.scad:

```
module ring(r1, r2, h) {
    difference() {
        cylinder(r = r1, h = h);
        translate([ 0, 0, -1 ]) cylinder(r = r2, h = h+2);
    }
}

ring(5, 4, 10);
```

Including the library using

```
include <ring.scad>;
rotate([90, 0, 0]) ring(10, 1, 1);
```

would result in the example ring being shown in addition to the rotated ring, but

```
use <ring.scad>;
rotate([90, 0, 0]) ring(10, 1, 1);
```

only shows the rotated ring.

If using the use function, make sure to place the use statements at top of the file, or at least not within a module!

This will work fine:

```
 // a.scad
 use <ring.scad>;
 module a() {
   ring();
 }
```

but this will result in an syntax error:

```
 //a.scad
 module a() {
   use <ring.scad>;
   ring();
 }
```

Default variables in an **`include`** can be overridden, for example

lib.scad

```
i=1;
k=3;
module x() {
    echo("hello world");
    echo("i=",i,"j=",j,"k=",k);
}
```

hello.scad

```
j=4;
include <lib.scad>;
x();
i=5;
x();
k=j;
x();
```

Produces the following

```
ECHO: "hello world"
ECHO: "i=", 5, "j=", 4, "k=", 4
ECHO: "hello world"
ECHO: "i=", 5, "j=", 4, "k=", 4
ECHO: "hello world"
ECHO: "i=", 5, "j=", 4, "k=", 4
```

However, placing **`j=4;`** after the **`include`** fails, producing

```
ECHO: "hello world"
ECHO: "i=", 5, "j=", 4, "k=", undef
ECHO: "hello world"
ECHO: "i=", 5, "j=", 4, "k=", undef
ECHO: "hello world"
ECHO: "i=", 5, "j=", 4, "k=", undef
```

Special variables

All variables starting with a '$' are special variables. The semantic is similar to the special variables in lisp: they have dynamic instead of lexical scoping.

What that means is that they're effectively automatically passed onward as arguments. Comparing a normal with a special variable:

```
normal=2;
module doesnt_pass_it()
{   echo(normal); }
module normal_mod()
{   doesnt_pass_it(); }
normal_mod(normal=1); //Should echo 2

$special=3; $another=5;
module passes_it()
{   echo($special, $another); }
module special_mod()
{   $another=6;
    passes_it();
}
special_mod($special=4); //Should echo 4,6
```

So basically it is useful when you do not want to pass many parameters all the time.

$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. Any attempt to set a lower value will cause a warning.

$fs is the minimum size of a fragment. Because of this variable very small circles have a smaller number of fragments than specified using $fa. The default value is 2. The minimum allowed value is 0.01. Any attempt to set a lower value will cause a warning.

$fn is usually 0. When this variable has a value greater than zero, the other two variables are ignored and full circle is rendered using this number of fragments. The default value is 0.

When $fa and $fs are used to determine the number of fragments for a circle, then OpenSCAD will never use 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));
}
```

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.

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);
```

$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 will be created in the same directory as the .scad file, using the following $t values, and saved in the following files:

- $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:

- $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, function of 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 will always output either Steps or Steps+1 files, though it may not be predictable which. When finished, it will wrap around and recreate each of the files, looping through and recreating them forever.

$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 will have 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.

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.

The OpenSCAD console supports a subset of HTML markup language. See here (http://qt-project.org/doc/qt-4.7/richtext-html-subset.html) for details.

It can be handy to use 'variable=variable' as the expression to easily label the variables, see the example below.

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);
//
echo("<b>Hello</b> <i>Qt!</i>");
```

Shows in the Console as

```
ECHO: "This is a cylinder with h=", 50, " and r=", 100
ECHO: my_h = 50, my_r = 100
ECHO: "Hello Qt!"
```

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);
}
```

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

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

[***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.

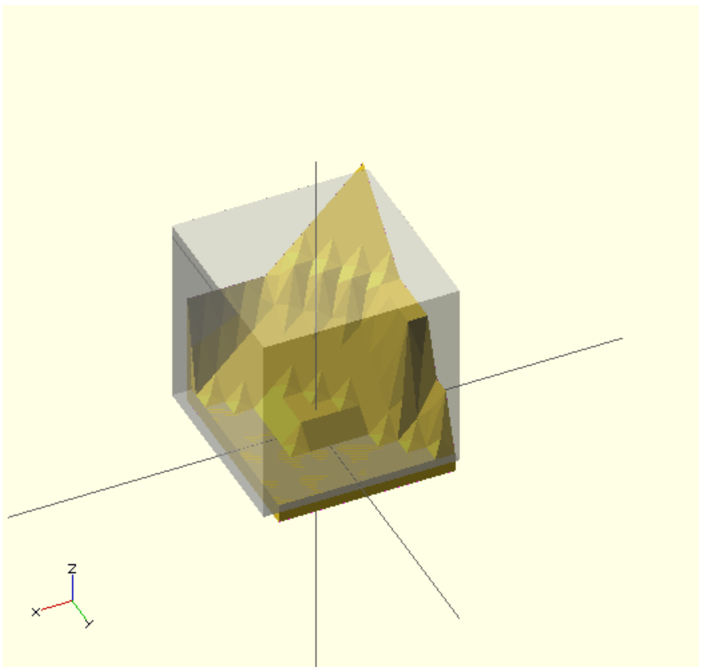
Examples

Example 1:

```
//surface.scad
surface(file = "surface.dat", center = true, convexity = 5);
%translate([0,0,5])cube([10,10,10], center =true);
```

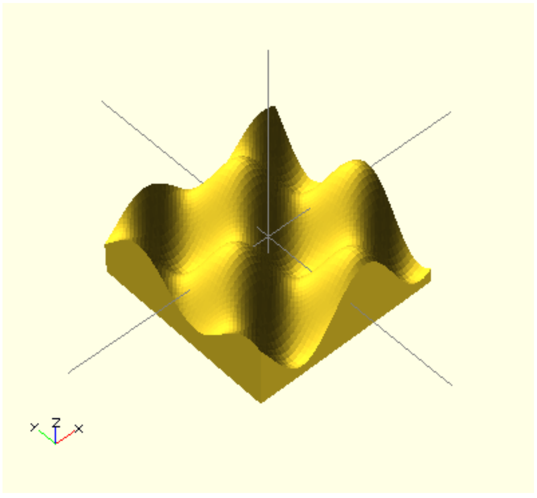
```
#surface.dat
10 9 8 7 6 5 5 5 5 5
9 8 7 6 6 4 3 2 1 0
8 7 6 6 4 3 2 1 0 0
7 6 6 4 3 2 1 0 0 0
6 6 4 3 2 1 1 0 0 0
6 6 3 2 1 1 1 0 0 0
6 6 2 1 1 1 1 0 0 0
6 6 1 0 0 0 0 0 0 0
3 1 0 0 0 0 0 0 0 0
3 0 0 0 0 0 0 0 0 0
```

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);
}
```

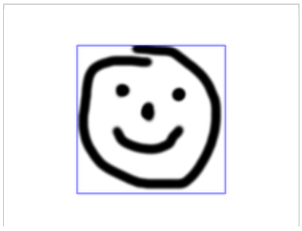


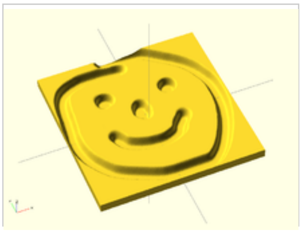
Example 3:

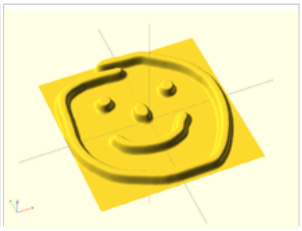
[***Note:*** *Requires version* ***2015.03***]

```
// 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);
```


Input image


Example 3a: surface(invert = false)


Example 3b: surface(invert = true)

Example 3: Using surface() with a PNG image as heightmap input.

Search

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

search(*match_value* , *string_or_vector* [, *num_returns_per_match* [, *index_col_num*]]);

Search Arguments

- **match_value**
  - Can be a single value or vector of values.
  - Strings are treated as vectors-of-characters to iterate over; the search function does **not** search for substrings.
  - **Note:** If *match_value* is a vector of strings, search will look for exact string matches.
    - See **Example 9** below.
- **string_or_vector**
  - The string or vector to search for matches.
- **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)
  - When *string_or_vector* is a vector-of-vectors, multidimensional table or more complex list-of-lists construct, the match_value may not be found in the first (index_col_num=0) column.
  - See **Example 5** below for a simple usage example.

Search Usage Examples

See **example023.scad** included with OpenSCAD for a renderable example.

Index values return as list

Example	Code	Result
1	`search("a","abcdabcd");`	[0]
2	`search("e","abcdabcd");`	[]
3	`search("a","abcdabcd",0);`	[[0,4]]
4	`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);`	[[0,4]] (see also Example 6 below)

Search on different column; return Index values

Example 5:

```
data= [ ["a",1],["b",2],["c",3],["d",4],["a",5],["b",6],["c",7],["d",8],["e",3] ];
search(3, data, num_returns_per_match=0, index_col_num=1);
```

Returns:

```
  [2,8]
```

Search on list of values

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

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

```
// 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
```

OpenSCAD Version

version() and version_num() will return OpenSCAD version number.

- The version() function will return the OpenSCAD version as a vector, e.g. [2011, 09, 23]
- The version_num() function will return the OpenSCAD version as a number, e.g. 20110923

parent_module(n) and $parent_modules

$parent_module 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 be used to e.g. build BOMs.

Example:

```
module top() {
  children();
}
module middle() {
  children();
}
top() middle() echo(parent_module(0)); // prints "middle"
top() middle() echo(parent_module(1)); // prints "top"
```

Using the 2D Subsystem

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.

square

Creates a square at the origin of the coordinate system. When center is true the square will be centered on the origin, otherwise it is created in the first quadrant. The argument names are optional if the arguments are given in the same order as specified in the parameters

Parameters

size
: Decimal or 2 value array. If a single number is given, the result will be a square with sides of that length. If a 2 value array is given, then the values will correspond to the lengths of the X and Y sides. Default value is 1.

center
: Boolean. This determines the positioning of the object. If true, object is centered at (0,0). Otherwise, the square is placed in the positive quadrant with one corner at (0,0). Defaults to false.

Example

```
square ([2,2],center = true);
```

circle

Creates a circle at the origin of the coordinate system. The argument name is optional.

Parameters

r
: Decimal. This is the radius of the circle. The resolution of the circle will be based on the size of the circle. If you need a small, high resolution circle you can get around this by making a large circle, then scaling it down by an appropriate factor, or you could set $fn or other special variables. Default value is 1.

d
: Decimal. This is the diameter of the circle. The resolution of the circle will be based on the size of the circle. If you need a small, high resolution circle you can get around this by making a large circle, then scaling it down by an appropriate factor, or you could set $fn or other special variables. Default value is 1.

(NOTE: d is only available in versions later than 2014.03. Debian is currently know to be behind this)

Examples

```
circle();  // uses default radius, r=1
```

```
circle(r = 10);
circle(d = 20);
```

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

polygon

Create a polygon with the specified points and paths.

Parameters

points
: vector of 2 element vectors, ie. the list of points of the polygon

paths
: Either a single vector, enumerating the point list, ie. the order to traverse the points, or, a vector of vectors, ie a list of point lists for each separate curve of the polygon. The latter is required if the polygon has holes. The parameter is optional and if omitted the points are assumed in order. (The 'pN' components of the *paths* vector are 0-indexed references to the elements of the *points* vector.)

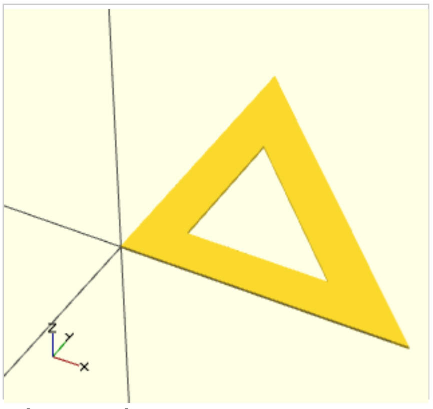
convexity
: Integer. Number of "inward" curves, ie. expected path crossings of an arbitrary line through the polygon. See below.

Usage

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

Example

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



Polygon example

In this example, we have 6 points (three for the "outer" triangle, and three for the "inner" one). We connect each one with two 2 path. In plain English, each element of a path must correspond to the position of a point defined in the points vector, e.g. "1" refers to [100,0].

Notice: In order to get a 3D object, you either extrude a 2D polygon (linear or (rotation) or directly use the polyhedron primitive solid. When using extrusion to form solids, its important to realize that the winding direction of the polygon is significant. If a polygon is wound in the wrong direction with respect to the axis of rotation, the final solid (after extrusion) may end up invisible. This problem can be checked for by flipping the polygon using scale([-1,1]) (assuming that extrusion is being done about the Z axis as it is by default).

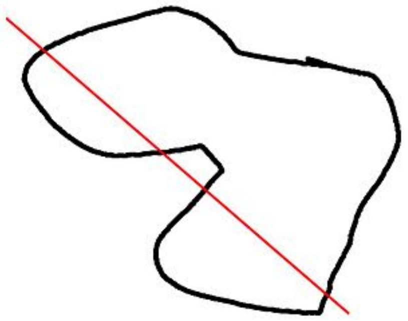
Notice: Althought the 2D drawing commands operate in axes labeled as X and Y, the extrusion commands implicitly translate these objects in X-Z coordinates and rotate about the Z axis.

Example:

```
 polygon([[0,0],[10,90],[11,-10]], convexity = N);
```

convexity

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.

import_dxf

DEPRECATED: The import_dxf() module will be removed in future releases. Use import() instead.

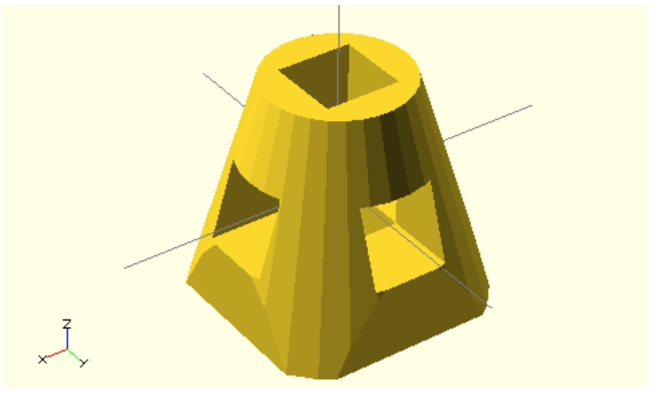
Read a DXF file and create a 2D shape.

Example

```
linear_extrude(height = 5, center = true, convexity = 10)
                import_dxf(file = "example009.dxf", layer = "plate");
```

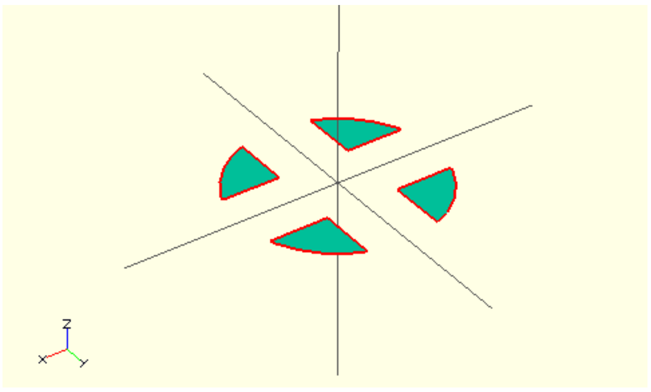
Using the `projection()` function, you can create 2d drawings from 3d models, and export them to the dxf format. It works by projecting a 3D model to the (x,y) plane, with z at 0. If `cut=true`, only points with z=0 will be considered (effectively cutting the object), with `cut=false`, points above and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

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



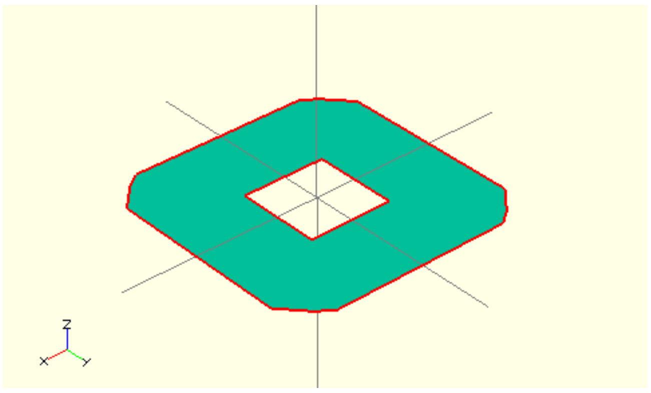
Then you can do a 'cut' projection, which gives you the 'slice' of the x-y plane with z=0.

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



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

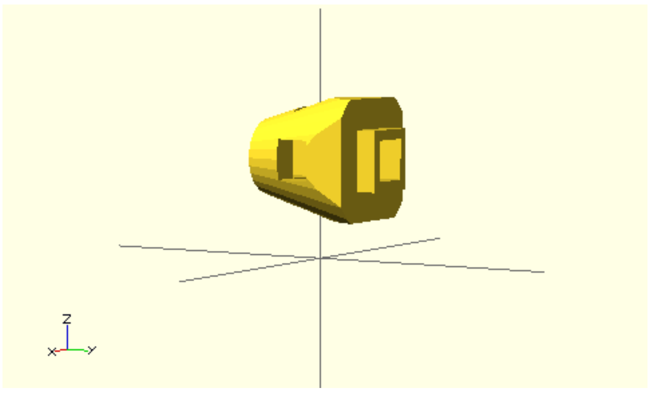
```
projection(cut = false) example002();
```



Another Example

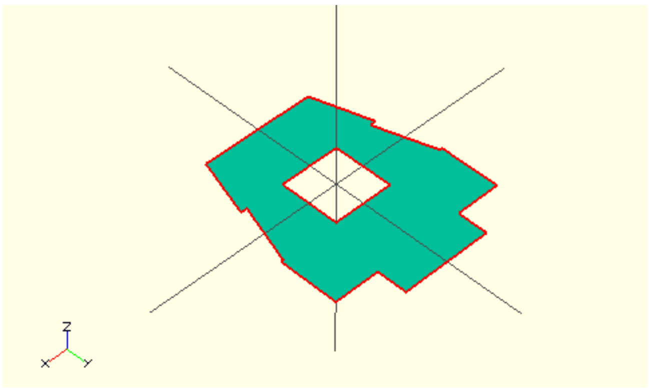
You can also use projection to get a 'side view' of an object. Let's take example002, and move it up, out of the X-Y plane, and rotate it:

```
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 (http://svn.clifford.at/openscad/trunk/examples/example021.scad).
- More complicated example (http://www.gilesbathgate.com/2010/06/extracting-2d-mendel-outlines-using-openscad/) from Giles Bathgate's blog

It is possible to use extrusion commands to convert 2D objects to 3D objects. This can be done with the built-in 2D primitives, like squares and circles, but also with arbitrary polygons.

Linear Extrude

Linear Extrusion is a modeling operation that takes a 2D polygon as input and extends it in the third dimension. This way a 3D shape is created.

Usage

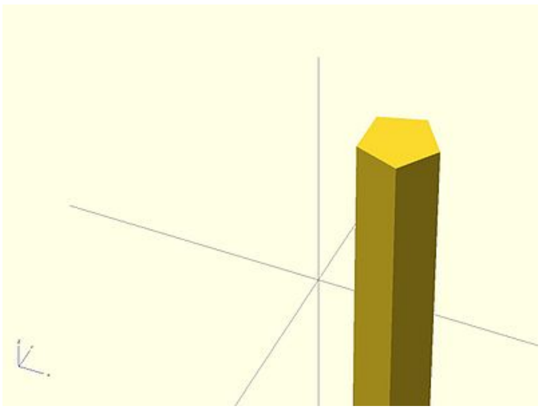
```
linear_extrude(height = fanwidth, center = true, convexity = 10, twist = -fanrot, slices = 20, scale = 1.0) {...}
```

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

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

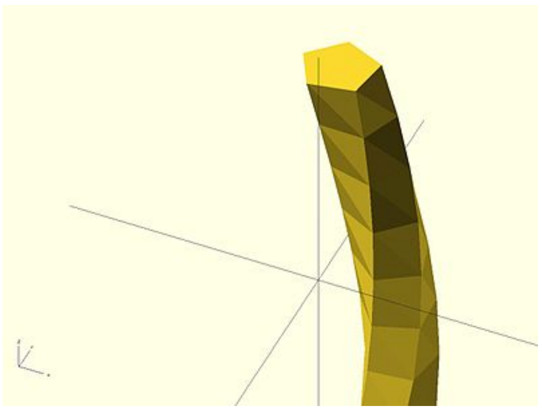
Twist

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



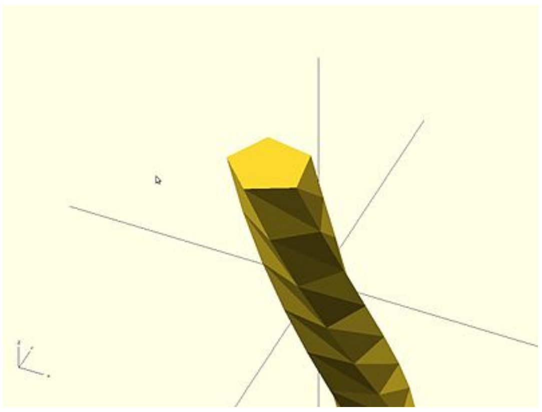
0° of Twist

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



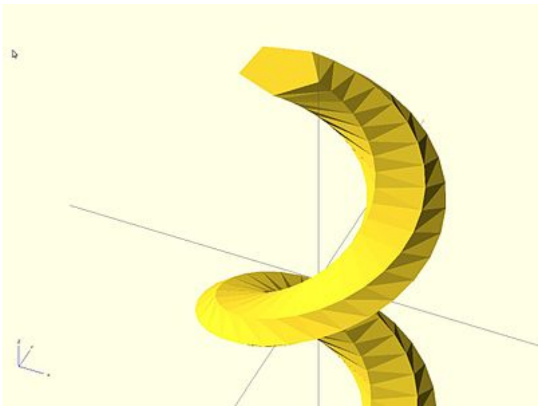
-100° of Twist

```
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);
```

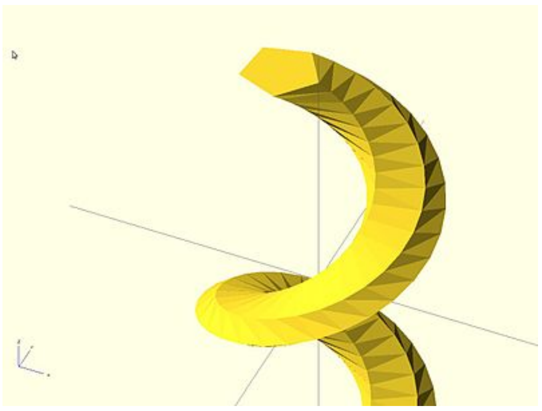


-500° of Twist

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

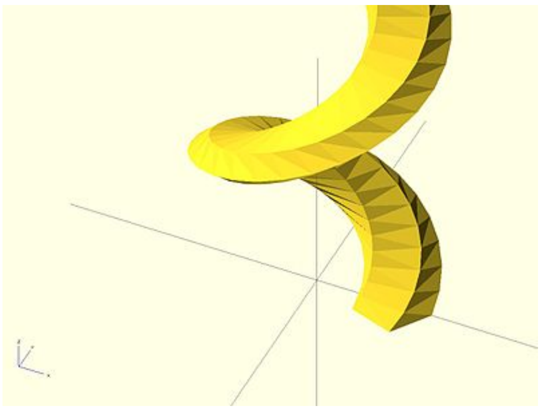
Center

Center determines if the object is centered on the Z-axis after extrusion. If it is set to false, the object would not center along Z-axis. See examples below.



center = true

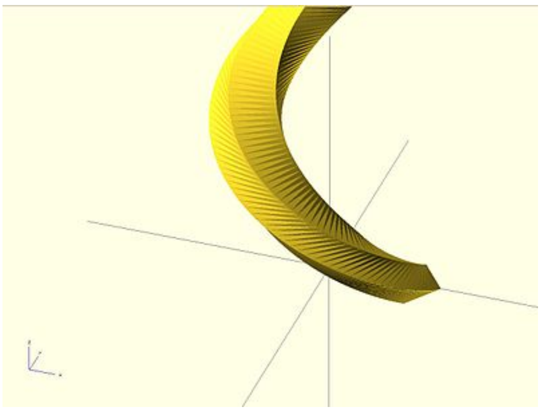
```
linear_extrude(height = 10, center = true, convexity = 10, twist = -500)
translate([2, 0, 0])
circle(r = 1);
```



center = false

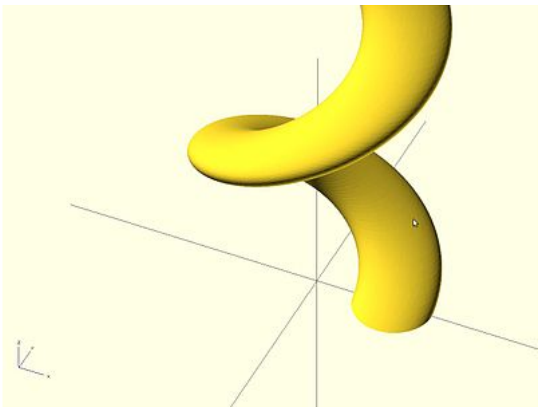
```
linear_extrude(height = 10, center = false, convexity = 10, twist = -500)
translate([2, 0, 0])
circle(r = 1);
```

Mesh Refinement



The slices parameter can be used to improve the output.

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



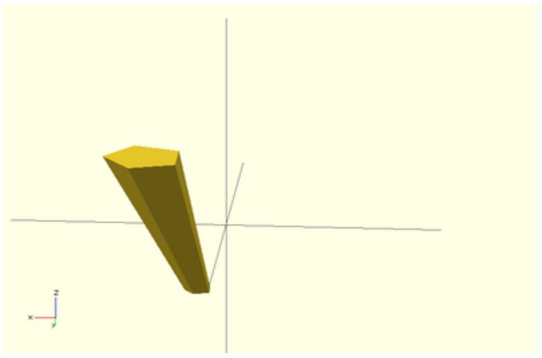
The special variables $fn, $fs and $fa can also be used to improve the output.

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

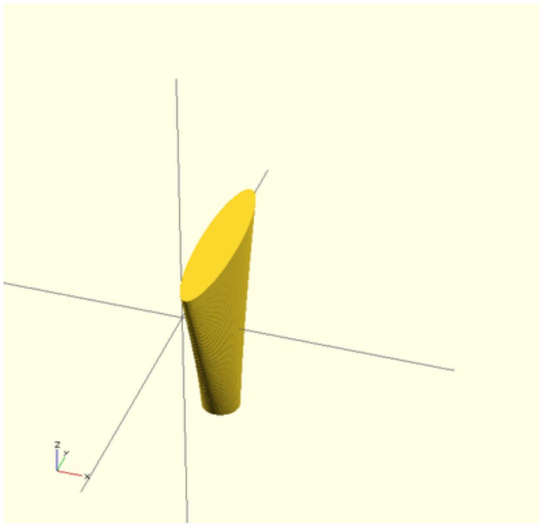
Scale

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 (https://github.com/openscad/openscad/issues/1341).

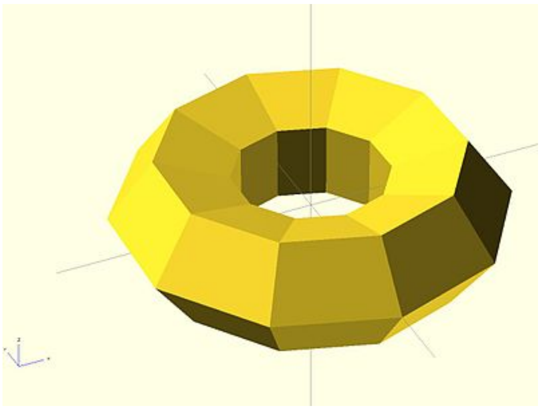
```
linear_extrude(height=10, scale=[1,0], slices=20, twist=0)
polygon(points=[[0,0],[20,10],[20,-10]]);
```

Rotate Extrude

A rotational extrusion is a Linear Extrusion with a twist, literally. Unfortunately, it can not be used to produce a helix for screw threads as the 2D outline must be normal to the axis of rotation, ie they need to be flat in 2D space.

The 2D shape needs to be either completely on the positive, or negative side (not recommended), of the X axis. It can touch the axis, i.e. zero, however if the shape crosses the X axis a warning will be shown in the console windows and the rotate_extrude() will be ignored. If the shape is in the negative axis the faces will be inside-out, you probably don't want to do that; it may be fixed in the future.

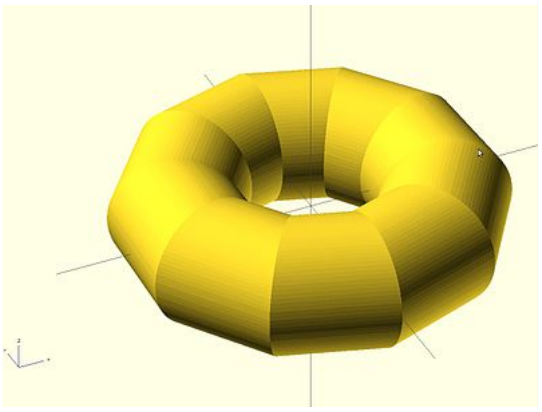
Examples



A simple torus can be constructed using a rotational extrude.

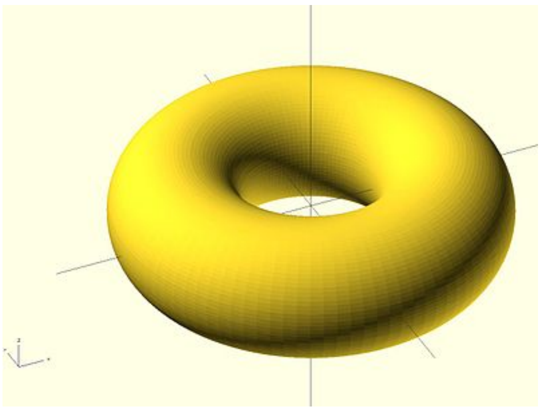
```
rotate_extrude(convexity = 10)
translate([2, 0, 0])
circle(r = 1);
```

Mesh Refinement



Increasing the number of fragments that the 2D shape is composed of will improve the quality of the mesh, but take 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.

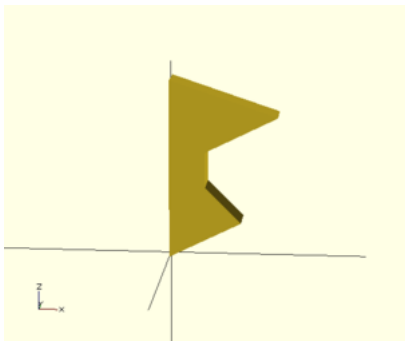
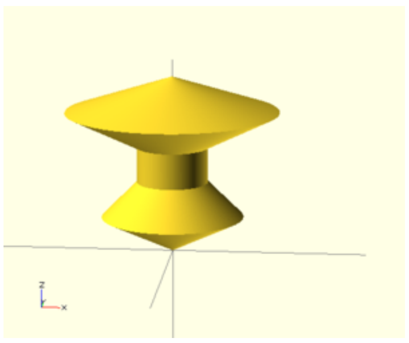
```
rotate_extrude(convexity = 10, $fn = 100)
translate([2, 0, 0])
circle(r = 1, $fn = 100);
```

Extruding a Polygon

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

Here is a simple polygon and its (fine-grained: `$fn=200`) rotational extrusion (profile and lathe). (Note it has been rotated 90 degrees to show how the rotation will look, 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]] );
```

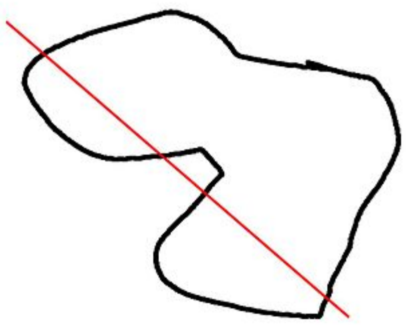



For more information on polygons, please see: 2D Primitives: Polygon.

Description of extrude parameters

Extrude parameters for all extrusion modes

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

height	The extrusion height
center	If true the solid will be 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.

With the import() and extrusion statements it is possible to convert 2D objects read from DXF files to 3D objects.

Linear Extrude

Example of linear extrusion of a 2D object imported from a DXF file.

```
linear_extrude(height = fanwidth, center = true, convexity = 10)
   import (file = "example009.dxf", layer = "fan_top");
```

Rotate Extrude

Example of rotational extrusion of a 2D object imported from a DXF file.

```
rotate_extrude(convexity = 10, twist = -fanrot)
   import (file = "example009.dxf", layer = "fan_side", origin = fan_side_center);
```

Getting Inkscape to work

Inkscape is an open source drawing program. Tutorials for transferring 2d DXF drawings from Inkscape to OpenSCAD are available here:

- http://repraprip.blogspot.com/2011/05/inkscape-to-openscad-dxf-tutorial.html (Very simple, needs path segments to be straight lines)
- http://tonybuser.com/?tag=inkscape (More complicated, involves conversion to Postscript)
- http://bobcookdev.com/inkscape/inkscape-dxf.html (Better DXF Export, native support for bezier curves)
- http://www.bigbluesaw.com/saw/big-blue-saw-blog/general-updates/big-blue-saws-dxf-export-for-inkscape.html (even better support, works as of 10/29/2014, see link below registration window. Note: As of 6/17/15 only works with version 0.48.5 or earlier of inkscape, due to a breaking change made in 0.91.)
- http://www.instructables.com/id/Convert-any-2D-image-to-a-3D-object-using-OpenSCAD/ (Convert any 2D image to a 3D object using OpenSCAD)

Description of extrude parameters

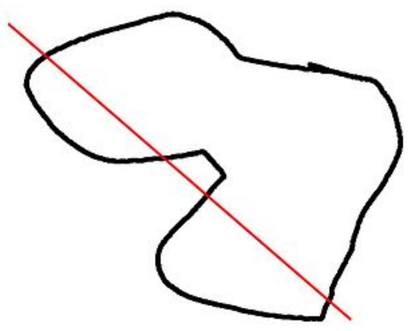
Extrude parameters for all extrusion modes

scale	FIXME
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. See diagram below.
file	The name of the DXF file to extrude [DEPRECATED]
layer	The name of the DXF layer to extrude [DEPRECATED]
origin	[x,y] coordinates to use as the drawing's center, in the units specified in the DXF file [DEPRECATED]

Extrude parameters for linear extrusion only

height	The extrusion height
center	If true, extrusion is half up and half down. If false, the section is extruded up.
twist	The extrusion twist in degrees
slices	FIXME

Convexity



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.

Currently, OpenSCAD only supports DXF as a graphics format for 2D graphics. Other common formats are PS/EPS, SVG and AI.

PS/EPS

The pstoedit (http://www.pstoedit.net/) program can convert between various vector graphics formats. OpenSCAD needs the `-polyaslines` option passed to the dxf output plugin to understand the file. The `-mm` option sets one mm to be one unit in the dxf; include this if you use one unit in OpenSCAD as equal to one millimeter. The `-dt` options instructs pstoedit to render texts, which is usually what you want if you include text. (If the rendered text's resolution in terms of polygon count is too low, the easiest solution is to scape up the eps before converting; if you know a more elegant solution, please add it to the example.)

```
pstoedit -dt -f "dxf: -polyaslines -mm" infile.eps outfile.dxf
```

SVG

Inkscape (http://inkscape.org) can convert SVG to EPS. Then pstoedit can convert the EPS to DXF.

```
inkscape -E intermediate.eps infile.svg
pstoedit -dt -f dxf:-polyaslines\ -mm intermediate.eps outfile.dxf
```

Makefile automation

The conversion can be automated using the make system; put the following lines in your `Makefile`:

```
all: my_first_file.dxf my_second_file.dxf another_file.dxf

%.eps: %.svg
	inkscape -E $@ $<
 %.dxf: %.eps
        pstoedit -dt -f dxf:-polyaslines\ -mm $< $@
```

The first line specifies which dxf files are to be generated when `make` is called in the current directory. The second paragraph specifies how to convert a file ending in .svg to a file ending in .eps, and the third from .eps to .dxf.

A more complete makefile could autogenerate dxf files from the any svg in the folder. In which case, put the following lines into your `Makefile`:

```
SVG := $(wildcard *.svg)
DXF := $(SVG:%.svg=%.dxf)
EPS := $(SVG:%.svg=%.eps)

.PHONY: all clean clean-eps clean-dxf

all: $(DXF)

%.eps: %.svg
        inkscape -E $*.eps $*.svg

%.dxf: %.eps
        pstoedit -dt -f "dxf: -polyaslines -mm" $*.eps $*.dxf

clean: clean-dxf clean-eps

clean-dxf:
        rm -f $(DXF)

clean-eps:
        rm -f $(EPS)
```

It's still possible to call `make filename.dxf` to build a particular file, but this code also allows for (re)building of all dxf files in a folder just by calling `make` or `make all`.

This code is also universal enough that it's possible to put the code in a single file and symlink every makefile in any directory that has svg files for dxf conversion by running:

```
ls -s /path/to/this/svg_to_dxf_makefile makefile
```

in each respective directory.

AI (Adobe Illustrator)

Although Adobe Illustrator CC/CC.2014 allows you to export illustrations as DXF (and select DXF format versions as early as 12), it will use DXF entities that are not supported by OpenSCAD, such as `POLYLINE` and `SPLINE`.

Since pstoedit does not natively support Adobe Illustrator files, one alternative is to use EXDXF (http://www.baby-universe.co.jp/en/plug-in/products/exdxf-pro/) which is an Adobe Illustrator plug-in (30 free trial exports and then you have to pay $90 to register the plugin).

Before exporting, it is recommended that you ensure that your Artboard is the same dimensions as the component you are exporting. Although EXDXF provides you with numerous options when exporting to DXF the most important option for OpenSCAD compliance is to set `Line Conversion` to `Line and Arc`.

OpenSCAD doesn't always provide information about the issues it encountered with a DXF import. If this happens, select `Design | Flush Caches` and then `Design | Reload and Compile`.

STL Import and Export

Import and Export

A prime ingredient of any 3D design flow is the ability to import from and export to other tools. The STL file format (http://en.wikipedia.org/wiki/STL_(file_format)) is currently the most common format used.

Import

import

Imports a file for use in the current OpenSCAD model. OpenSCAD currently supports import of DXF and STL (both ASCII and Binary) files.

Parameters

<file>
: A string containing the path to the STL or DXF file.

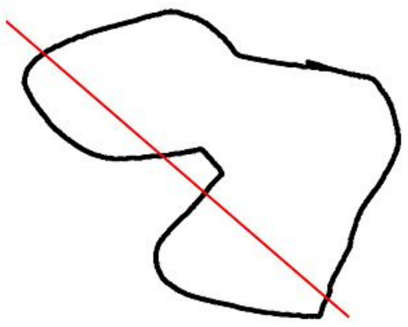
<convexity>
: An Integer. The convexity parameter specifies the maximum number of front sides (back sides) a ray intersecting the object might penetrate. This parameter is only needed for correctly displaying the object in OpenCSG preview mode and has no effect on the polyhedron rendering.

Usage examples:

```
import("example012.stl", convexity=3);
import("D:\\Documents and Settings\\User\\My Documents\\Gear.stl", convexity=3);
```

(Windows users must "escape" the backslashes by writing them doubled.)

Convexity



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.

Notes

In the latest version of OpenSCAD, import() is now used for importing both 2D (DXF for extrusion) and 3D (STL) files.

If you want to render the imported STL file later, you have to make sure that the STL file is "clean". This means that the mesh has to be manifold and should not contain holes nor self-intersections. If the STL is not clean, you might get errors like:

```
CGAL error in CGAL_Build_PolySet: CGAL ERROR: assertion violation!
Expr: check_protocoll == 0
File: /home/don/openscad_deps/mxe/usr/i686-pc-mingw32/include/CGAL/Polyhedron_incremental_builder_3.h
Line: 199
```

or

```
CGAL error in CGAL_Nef_polyhedron3(): CGAL ERROR: assertion violation!
Expr: pe_prev->is_border() || !internal::Plane_constructor<Plane>::get_plane(pe_prev->facet(),pe_prev->facet()->plane()).is_degenerate()
File: /home/don/openscad_deps/mxe/usr/i686-pc-mingw32/include/CGAL/Nef_3/polyhedron_3_to_nef_3.h
Line: 253
```

In order to clean the STL file, you have the following options:

- use http://wiki.netfabb.com/Semi-Automatic_Repair_Options . This will repair the holes but not the self-intersections.

- use netfabb basic. This free software doesnt have the option to close holes nor can it fix the self-intersections

- use MeshLab, This free software can fix all the issues

Using MeshLab, you can do:

- Render - Show non Manif Edges

- Render - Show non Manif Vertices

- if found, use Filters - Selection - Select non Manifold Edges or Select non Manifold Vertices - Apply - Close. Then click button 'Delete the current set of selected vertices...' or check http://www.youtube.com/watch?v=oDx0Tgy0UHo for an instruction video. The screen should show "0 non manifold edges", "0 non manifold vertices"

Next, you can click the icon 'Fill Hole', select all the holes and click Fill and then Accept. You might have to redo this action a few times.

Use File - Export Mesh to save the STL.

import_stl

<DEPRECATED.. Use the command **import** instead..>

Imports an STL file for use in the current OpenSCAD model

Parameters

<file>
A string containing the path to the STL file to include.

<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.

Usage examples:

```
import_stl("example012.stl", convexity = 5);
```

STL Export

STL Export

To export your design, select "Export as STL..." from the "File --> Export" menu, then enter a filename in the ensuing dialog box. Don't forget to add the ".stl" extension.

Trouble shooting:

After *compile and render GCAL* (F6), you may see that your design is *simple: no*. That's bad news.

See line 8 in the following output from *OpenSCAD 2010.02*:

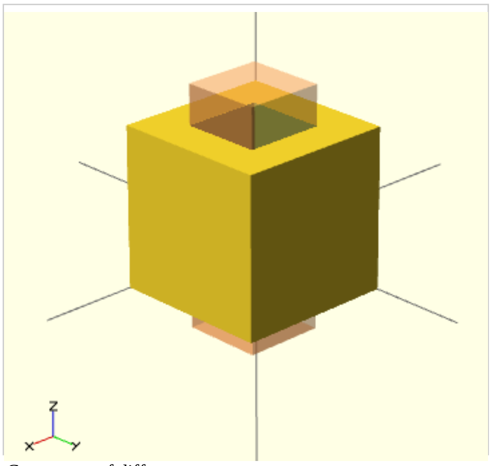
```
Parsing design (AST generation)...
Compiling design (CSG Tree generation)...
Compilation finished.
Rendering Polygon Mesh using CGAL...
Number of vertices currently in CGAL cache: 732
Number of objects currently in CGAL cache: 12
   Top level object is a 3D object:
   Simple:         no                <*****************
   Valid:         yes
   Vertices:       22
   Halfedges:      70
   Edges:          35
   Halffacets:     32
   Facets:         16
   Volumes:         2
Total rendering time: 0 hours, 0 minutes, 0 seconds
Rendering finished.
```

When you try to export this to .STL you will get a message like:

```
Object isn't a valid 2-manifold! Modify your design..
```

"Manifold" means that it is "water tight" and that there are no holes in the geometry. In a valid 2-manifold each edge must connect exactly two facets. That means that the program must be able to connect a face with an object. E.g. if you use a cube of height 10 to carve out something from a wider cube of height 10, it is not clear to which cube the top or the bottom belongs. So make the small extracting cube a bit "longer" (or "shorter"):

```
difference() {
        // original
        cube (size = [2,2,2]);
        // object that carves out
        # translate ([0.5,0.5,-0.5]) {
            cube (size = [1,1,3]);
        }
}
```

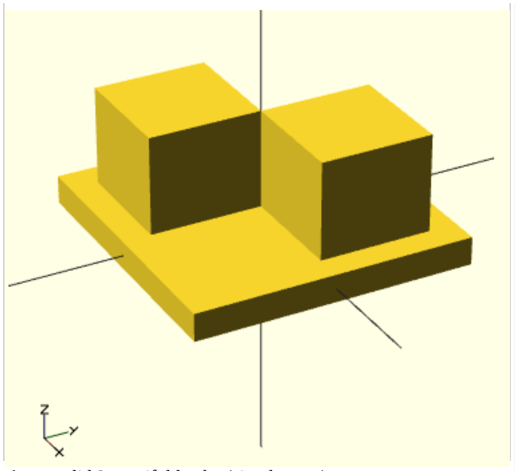


Correct use of difference

Here is a more tricky little example taken from the OpenSCAD (http://rocklinux.net/pipermail/openscad/2009-December/000018.html) Forum (retrieved 15:13, 22 March 2010 (UTC)):

```
module example1() {
                cube([20, 20, 20]);
                translate([-20, -20, 0]) cube([20, 20, 20]);
                cube([50, 50, 5], center = true);
        }
module example2() {
                cube([20.1, 20.1, 20]);
                translate([-20, -20, 0]) cube([20.1, 20.1, 20]);
                cube([50, 50, 5], center = true);
        }
```

Example1 would render like this:



A not valid 2-manifold cube (simple = no)

The **example1** module is not a valid 2-manifold because both cubes are sharing one edge. They touch each other but do not intersect.

Example2 is a valid 2-manifold because there is an intersection. Now the construct meets the 2-manifold constraint stipulating that *each edge* must connect exactly two facets.

Pieces you are subtracting must extend past the original part. (OpenSCAD Tip: Manifold Space and Time (http://www.iheartrobotics.com/2010/01/openscad-tip-manifold-space-and-time.html), retrieved 18:40, 22 March 2010 (UTC)).

For reference, another situation that causes the design to be non-exportable is when two faces that are each the result of a subtraction touch. Then the error message comes up.

```
difference () {
   cube ([20,10,10]);
   translate ([10,0,0]) cube (10);
}
difference () {
   cube ([20,10,10]);
   cube (10);
}
```

simply touching surfaces is correctly handled.

```
translate ([10,0,0]) cube (10);
cube (10);
```

import

Imports a file for use in the current OpenSCAD model. OpenSCAD currently supports import of DXF and STL (both ASCII and Binary) files.

Parameters

<file>
: A string containing the path to the STL or DXF file.

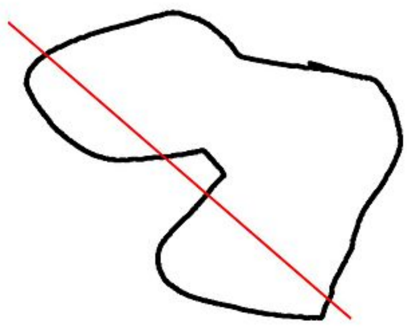
<convexity>
: An Integer. The convexity parameter specifies the maximum number of front sides (back sides) a ray intersecting the object might penetrate. This parameter is only needed for correctly displaying the object in OpenCSG preview mode and has no effect on the polyhedron rendering.

Usage examples:

```
import("example012.stl", convexity=3);
import("D:\\Documents and Settings\\User\\My Documents\\Gear.stl", convexity=3);
```

(Windows users must "escape" the backslashes by writing them doubled.)

Convexity



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.

Notes

In the latest version of OpenSCAD, import() is now used for importing both 2D (DXF for extrusion) and 3D (STL) files.

If you want to render the imported STL file later, you have to make sure that the STL file is "clean". This means that the mesh has to be manifold and should not contain holes nor self-intersections. If the STL is not clean, you might get errors like:

```
CGAL error in CGAL_Build_PolySet: CGAL ERROR: assertion violation!
Expr: check_protocoll == 0
File: /home/don/openscad_deps/mxe/usr/i686-pc-mingw32/include/CGAL/Polyhedron_incremental_builder_3.h
Line: 199
```

or

```
CGAL error in CGAL_Nef_polyhedron3(): CGAL ERROR: assertion violation!
Expr: pe_prev->is_border() || !internal::Plane_constructor<Plane>::get_plane(pe_prev->facet(),pe_prev->facet()->plane()).is_degenerate()
File: /home/don/openscad_deps/mxe/usr/i686-pc-mingw32/include/CGAL/Nef_3/polyhedron_3_to_nef_3.h
Line: 253
```

In order to clean the STL file, you have the following options:

- use http://wiki.netfabb.com/Semi-Automatic_Repair_Options . This will repair the holes but not the self-intersections.

- use netfabb basic. This free software doesnt have the option to close holes nor can it fix the self-intersections

- use MeshLab, This free software can fix all the issues

Using MeshLab, you can do:

- Render - Show non Manif Edges

- Render - Show non Manif Vertices

- if found, use Filters - Selection - Select non Manifold Edges or Select non Manifold Vertices - Apply - Close. Then click button 'Delete the current set of selected vertices...' or check http://www.youtube.com/watch?v=oDx0Tgy0UHo for an instruction video. The screen should show "0 non manifold edges", "0 non manifold vertices"

Next, you can click the icon 'Fill Hole', select all the holes and click Fill and then Accept. You might have to redo this action a few times.

Use File - Export Mesh to save the STL.

import_stl

<DEPRECATED.. Use the command **import** instead..>

Imports an STL file for use in the current OpenSCAD model

Parameters

<file>
A string containing the path to the STL file to include.

<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.

Usage examples:

```
import_stl("example012.stl", convexity = 5);
```

STL Export

To export your design, select "Export as STL..." from the "File --> Export" menu, then enter a filename in the ensuing dialog box. Don't forget to add the ".stl" extension.

Trouble shooting:

After *compile and render GCAL* (F6), you may see that your design is *simple: no*. That's bad news.

See line 8 in the following output from *OpenSCAD 2010.02*:

```
Parsing design (AST generation)...
Compiling design (CSG Tree generation)...
Compilation finished.
Rendering Polygon Mesh using CGAL...
Number of vertices currently in CGAL cache: 732
Number of objects currently in CGAL cache: 12
   Top level object is a 3D object:
```

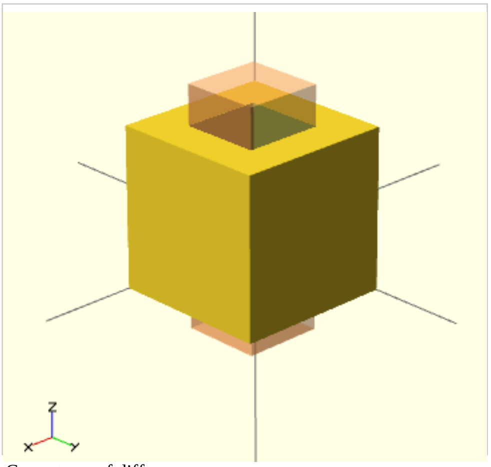
```
   Valid:         yes
   Vertices:       22
   Halfedges:      70
   Edges:          35
   Halffacets:     32
   Facets:         16
   Volumes:         2
Total rendering time: 0 hours, 0 minutes, 0 seconds
Rendering finished.
```

When you try to export this to .STL you will get a message like:

```
Object isn't a valid 2-manifold! Modify your design..
```

"Manifold" means that it is "water tight" and that there are no holes in the geometry. In a valid 2-manifold each edge must connect exactly two facets. That means that the program must be able to connect a face with an object. E.g. if you use a cube of height 10 to carve out something from a wider cube of height 10, it is not clear to which cube the top or the bottom belongs. So make the small extracting cube a bit "longer" (or "shorter"):

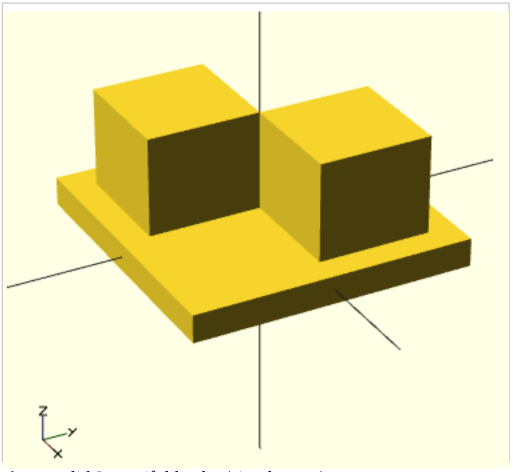
```
difference() {
        // original
        cube (size = [2,2,2]);
        // object that carves out
        # translate ([0.5,0.5,-0.5]) {
            cube (size = [1,1,3]);
        }
}
```


Correct use of difference

Here is a more tricky little example taken from the OpenSCAD (http://rocklinux.net/pipermail/openscad/2009-December/000018.html) Forum (retrieved 15:13, 22 March 2010 (UTC)):

```
module example1() {
                cube([20, 20, 20]);
                translate([-20, -20, 0]) cube([20, 20, 20]);
                cube([50, 50, 5], center = true);
        }
module example2() {
                cube([20.1, 20.1, 20]);
                translate([-20, -20, 0]) cube([20.1, 20.1, 20]);
                cube([50, 50, 5], center = true);
        }
```

Example1 would render like this:


A not valid 2-manifold cube (simple = no)

The **example1** module is not a valid 2-manifold because both cubes are sharing one edge. They touch each other but do not intersect.

Example2 is a valid 2-manifold because there is an intersection. Now the construct meets the 2-manifold constraint stipulating that *each edge* must connect exactly two facets.

Pieces you are subtracting must extend past the original part. (OpenSCAD Tip: Manifold Space and Time (http://www.iheartrobotics.com/2010/01/openscad-tip-manifold-space-and-time.html), retrieved 18:40, 22 March 2010 (UTC)).

For reference, another situation that causes the design to be non-exportable is when two faces that are each the result of a subtraction touch. Then the error message comes up.

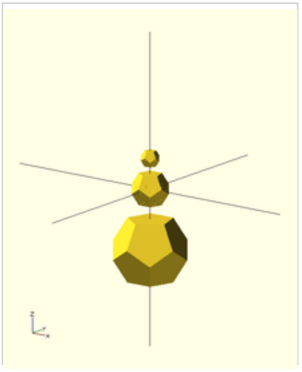
```
difference () {
   cube ([20,10,10]);
   translate ([10,0,0]) cube (10);
}
difference () {
   cube ([20,10,10]);
   cube (10);
}
```

simply touching surfaces is correctly handled.

```
translate ([10,0,0]) cube (10);
cube (10);
```

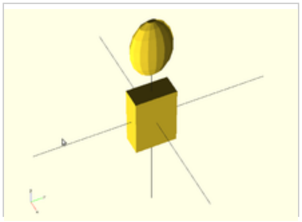
Dodecahedron

```
//create a dodecahedron by intersecting 6 boxes
module dodecahedron(height)
{
        scale([height,height,height]) //scale by height parameter
        {
                intersection(){
                        //make a cube
                        cube([2,2,1], center = true);
                        intersection_for(i=[0:4]) //loop i from 0 to 4, and intersect results
                        {
                                //make a cube, rotate it 116.565 degrees around the X axis,
                                //then 72*i around the Z axis
                                rotate([0,0,72*i])
                                        rotate([116.565,0,0])
                                        cube([2,2,1], center = true);
                        }
                }
        }
}
//create 3 stacked dodecahedra
//call the module with a height of 1 and move up 2
translate([0,0,2])dodecahedron(1);
//call the module with a height of 2
dodecahedron(2);
//call the module with a height of 4 and move down 4
translate([0,0,-4])dodecahedron(4);
```


The Dodecahedron as rendered from the example.

Bounding Box

```
// Rather kludgy module for determining bounding box from intersecting projections
module BoundingBox()
{
        intersection()
        {
                translate([0,0,0])
                linear_extrude(height = 1000, center = true, convexity = 10, twist = 0)
                projection(cut=false) intersection()
                {
                        rotate([0,90,0])
                        linear_extrude(height = 1000, center = true, convexity = 10, twist = 0)
                        projection(cut=false)
                        rotate([0,-90,0])
                        children(0);

                        rotate([90,0,0])
                        linear_extrude(height = 1000, center = true, convexity = 10, twist = 0)
                        projection(cut=false)
                        rotate([-90,0,0])
                        children(0);
                }
                rotate([90,0,0])
                linear_extrude(height = 1000, center = true, convexity = 10, twist = 0)
                projection(cut=false)
                rotate([-90,0,0])
                intersection()
                {
                        rotate([0,90,0])
                        linear_extrude(height = 1000, center = true, convexity = 10, twist = 0)
                        projection(cut=false)
                        rotate([0,-90,0])
                        children(0);

                        rotate([0,0,0])
                        linear_extrude(height = 1000, center = true, convexity = 10, twist = 0)
                        projection(cut=false)
                        rotate([0,0,0])
                        children(0);
                }
        }
}

// Test module on ellipsoid
translate([0,0,40]) scale([1,2,3]) sphere(r=5);
BoundingBox() scale([1,2,3]) sphere(r=5);
```


Bounding Box applied to an Ellipsoid

Command line usage

OpenSCAD can not only be used as a GUI, but also handles command line arguments. Its usage line says:

OpenSCAD 2014.03+ has these options:

```
openscad     [ -o output_file [ -d deps_file ] ]\
             [ -m make_command ] [ -D var=val [..] ] \
             [ --version ] [ --info ] \
             [ --camera=translatex,y,z,rotx,y,z,dist | \
               --camera=eyex,y,z,centerx,y,z ] \
             [ --imgsize=width,height ] [ --projection=(o)rtho|(p)ersp] \
             [ --render | --preview[=throwntogether] ] \
             [ --csglimit=num ] \
             filename
```

Openscad 2013.05 had these options:

```
openscad     [ -o output_file [ -d deps_file ] ]\
             [ -m make_command ] [ -D var=val [..] ] [ --render ] \
             [ --camera=translatex,y,z,rotx,y,z,dist | \
               --camera=eyex,y,z,centerx,y,z ] \
             [ --imgsize=width,height ] [ --projection=(o)rtho|(p)ersp] \
             filename
```

Earlier releases had only these:

```
openscad [ -o output_file [ -d deps_file ] ] \
         [ -m make_command ] [ -D var=val [..] ] filename
```

The usage on OpenSCAD version 2011.09.30 (now deprecated) was:

```
openscad [ { -s stl_file | -o off_file | -x dxf_file } [ -d deps_file ] ]\
         [ -m make_command ] [ -D var=val [..] ] filename
```

Export options

When called with the `-o` option, OpenSCAD will not start the GUI, but execute the given file and export the to the *output_file* in a format depending on the extension (`.stl` / `.off` / `.dxf`, `.csg`).

Some versions use -s/-d/-o to determine the output file format instead; check with "openscad --help".

If the option `-d` is given in addition to an export command, all files accessed while building the mesh are written in the argument of `-d` in the syntax of a Makefile.

Camera and image output

For 2013.05+, the option to output a `.png` image was added. There are two types of cameras available for the generation of images.

The first camera type is a 'gimbal' camera that uses Euler angles, translation, and a camera distance, like OpenSCAD's GUI viewport display at the bottom of the OpenSCAD window.

!!! There is a bug in the implementation of cmdline camera, where the rotations do not match the numbers in the GUI. This will be fixed in an upcoming release so that the GUI and cmdline camera variables will work identically.

The second camera type is a 'vector' camera, with an 'eye' camera location vector and a 'lookat' center vector.

--imgsize chooses the .png dimensions and --projection chooses orthogonal or perspective, as in the GUI.

By default, cmdline .png output uses Preview mode (f5) with OpenCSG. For some situations it will be desirable instead to use the full render, with CGAL. This is done by adding '--render' as an option.

Constants

In order to pre-define variables, use the `-D` option. It can be given repeatedly. Each occurrence of `-D` must be followed by an assignment. Unlike normal OpenSCAD assignments, these assignments don't define variables, but constants, which can not be changed inside the program, and can thus be used to overwrite values defined in the program at export time.

If you want to assign the -D variable to another variable, the -D variable MUST be initialised in the main .scad program

```
param1=0;   // must be initalised
len=param1; // param1 passed via -D on cmd-line
echo(len,param);
```

without the first line len wound be undefined.

The right hand sides can be arbitrary OpenSCAD expressions, including mathematical operations and strings. Be aware that strings have to be enclosed in quotes, which have to be escaped from the shell. To render a model that takes a quality parameter with the value "production", one has to run

```
openscad -o my_model_production.stl -D 'quality="production"' my_model.scad
```

Command to build required files

In a complex build process, some files required by an OpenSCAD file might be currently missing, but can be generated, for example if they are defined in a Makefile. If OpenSCAD is given the option `-m make`, it will start `make` *`file`* the first time it tries to access a missing *file*.

Makefile example

The `-d` and `-m` options only make sense together. (`-m` without `-d` would not consider modified dependencies when building exports, `-d` without `-m` would require the files to be already built for the first run that generates the dependencies.)

Here is an example of a basic Makefile that creates an .stl file from an .scad file of the same name:

```
# explicit wildcard expansion suppresses errors when no files are found
include $(wildcard *.deps)

%.stl: %.scad
        openscad -m make -o $@ -d $@.deps $<
```

When `make my_example.stl` is run for the first time, it finds no .deps files, and will just depend on `my_example.scad`; since `my_example.stl` is not yet preset, it will be created unconditionally. If OpenSCAD finds missing files, it will call `make` to build them, and it will list all used files in `my_example.stl.deps`.

When `make my_example.stl` is called subsequently, it will find and include `my_example.stl.deps` and check if any of the files listed there, including `my_example.scad`, changed since `my_example.stl` was built, based on their time stamps. Only if that is the case, it will build `my_example.stl` again.

Automatic targets

When building similar .stl files from a single .scad file, there is a way to automate that too:

```
# match "module foobar() { // `make` me"
TARGETS=$(shell sed '/^module [a-z0-9_-]*().*make..\?me.*$$/!d;s/module //;s/().*/.stl/' base.scad)

all: ${TARGETS}

# auto-generated .scad files with .deps make make re-build always. keeping the
# scad files solves this problem. (explanations are welcome.)
.SECONDARY: $(shell echo "${TARGETS}" | sed 's/\.stl/.scad/g')

# explicit wildcard expansion suppresses errors when no files are found
include $(wildcard *.deps)

%.scad:
        echo -n 'use <base.scad>\n$*();' > $@

%.stl: %.scad
        openscad -m make -o $@ -d $@.deps $<
```

All objects that are supposed to be exported automatically have to be defined in `base.scad` in an own module with their future file name (without the ".stl"), and have a comment like "`// make me`" in the line of the module definition. The "`TARGETS=`" line picks these out of the base file and creates the file names. These will be built when `make all` (or `make`, for short) is called.

As the convention from the last example is to create the .stl files from .scad files of the same base name, for each of these files, an .scad file has to be generated. This is done in the "`%.scad:`" paragraph; `my_example.scad` will be a very simple OpenSCAD file:

```
use <base.scad>
my_example();
```

The "`.SECONDARY`" line is there to keep `make` from deleting the generated .scad files. If it deleted it, it would not be able to automatically determine which files need no rebuild any more; please post ideas about what exactly goes wrong there (or how to fix it better) on the talk page!

Windows notes

On Windows, openscad.com should be called from the command line as a wrapper for openscad.exe. This is because Openscad uses the 'devenv' solution to the Command-Line/GUI output issue. Typing 'openscad' at the cmd.exe prompt will, by default, call the .com program wrapper.

MacOS notes

On MacOS the binary is normally hidden inside the App folder. If OpenSCAD is installed in the global Applications folder, it can be called from command line like in the following example that just shows the OpenSCAD version:

```
macbook:/$ /Applications/OpenSCAD.app/Contents/MacOS/OpenSCAD -v
OpenSCAD version 2013.06
```

Building OpenSCAD from Sources

Prebuilt binary packages

If you are lucky, you won't have to build it. Many Linux and BSD systems have pre-built OpenSCAD packages including Debian, Ubuntu, Fedora, Arch, NetBSD and OpenBSD. Check your system's package manager for details.

For Ubuntu systems you can also try chrysn's Ubuntu packages at his launchpad PPA (https://launchpad.net/~chrysn/+archive/openscad), or you can just copy/paste the following onto the command line:

```
sudo add-apt-repository ppa:chrysn/openscad
sudo apt-get update
sudo apt-get install openscad
```

His repositories for OpenSCAD and OpenCSG are here (http://archive.amsuess.com/pool/contrib/o/openscad/) and here (http://archive.amsuess.com/pool/main/o/opencsg/).

There is also a generic linux binary package at http://www.openscad.org that can be unpacked and run from within most linux systems. It is self contained and includes the required libraries.

Building OpenSCAD yourself

If you wish to build OpenSCAD for yourself, start by installing ***git*** on your system using your package manager. Git is sometimes packaged under the name 'scmgit' or 'git-core'. Then, use git to download the OpenSCAD source code

```
cd ~/
git clone https://github.com/openscad/openscad.git
cd openscad
```

Then get the MCAD library, which is now included with OpenSCAD binary distributions

```
git submodule init
git submodule update
```

Installing dependencies

OpenSCAD uses a large number of third-party libraries and tools. These are called dependencies. An up to date list of dependencies can usually be found in the README.md in openscad's main directory, here: https://github.com/openscad/openscad/ A brief list follows:

Eigen, GCC or Clang, Bison, Flex, CGAL, Qt, GMP, MPFR, boost, cmake, OpenCSG, GLEW, QScintilla, glib2, harfbuzz, freetype2, pkg-config, fontconfig

Prepackaged dependencies

Most systems are set up to install pre-built dependencies using a 'package manager', such as ***apt*** on ubuntu or ***pacman*** on Arch Linux. OpenSCAD comes with a 'helper script' that will attempt to automatically run your package manager for you and download and install these pre-built packages if they exist. Note you must be running as root and/or using *sudo* to try this. Note that these scripts will likely fail on Sun, Solaris, AIX, IRIX, etc (skip to the 'building dependencies' section below).

```
./scripts/uni-get-dependencies.sh
```

Verifying dependencies

After attempting to install dependencies, you should double check them. Exit any shells and perhaps reboot.

Now verify that the version numbers are up to those listed in openscad/README.md file. Also verify that no packages were accidentally missed. For example open a shell and run 'flex --version' or 'gcc --version'. These are good sanity checks to make sure your environment is proper.

OpenSCAD comes with another helper script that attempts to automate this process on many Linux and BSD systems (Again, it won't work on Sun/Solaris/Irix/AIX/etc).

```
./scripts/check-dependencies.sh
```

If you cannot verify that your dependencies are installed properly and of a sufficient version, then you may have to install some 'by hand' (see the section below on building your own dependencies).

If your system has all the proper versions of dependencies, then continue to the 'Building OpenSCAD' section.

Building the dependencies yourself

On systems that lack updated dependency libraries or tools, you will need to download each and build it and install it by hand. You can do this by downloading and following installation instructions for each package separately. However OpenSCAD comes with scripts that attempt to automate this process on Linux and BSD systems, by installing everything into a folder created under $HOME/openscad_deps. This script will not build typical development dependencies like X11, Qt4, gcc, bash etc. But it will attempt things like OpenCSG, CGAL, boost, etc.

To run the automated script, first set up the environment variables (if you don't use bash, replace "source" with a single ".")

```
source scripts/setenv-unibuild.sh
```

Then, run a second script to download and build.

```
./scripts/uni-build-dependencies.sh
```

(If you only need CGAL or OpenCSG, you can just run ' ./scripts/uni-build-dependencies.sh cgal' or opencsg and it will only build a single library.)

The complete download and build process can take several hours, depending on your network connection speed and system speed. It is recommended to have at least 2 Gigabyte of free disk space to do the full dependency build. Each time you log into a new shell and wish to re-compile OpenSCAD you need to re-run the 'source scripts/setenv-unibuild.sh' script.

After completion, return to the section above on 'verifying dependencies' to see if they installed correctly.

Build the OpenSCAD binary

Once you have installed your dependencies, you can build OpenSCAD.

```
qmake       # or qmake-qt4, depending on your distribution
make
```

You can also install OpenSCAD to /usr/local/ if you wish. The 'openscad' binary will be put under /usr/local/bin, the libraries and examples will be under something like /usr/local/share/openscad possibly depending on your system. Note that if you have previously installed a binary linux package of openscad, you should take care to delete /usr/local/lib/openscad and /usr/local/share/openscad because they are not the same paths as what the standard qmake-built 'install' target uses.

```
sudo make install
```

Note: on Debian-based systems create a package and install OpenSCAD using:

```
sudo checkinstall -D make install
```

If you prefer not to install you can run "`./openscad`" directly whilst still in the ~/openscad directory.

Compiling the test suite

OpenSCAD comes with over 740 regression tests. To build and run them, it is recommended to first build the GUI version of OpenSCAD by following the steps above, including the downloading of MCAD. Then, from the same login, run these commands:

```
cd tests
mkdir build && cd build
cmake ..
make
ctest -C All
```

The file 'openscad/doc/testing.txt' has more information. Full test logs are under `tests/build/Testing/Temporary`. A pretty-printed index.html web view of the tests can be found under a machine-specific subdirectory thereof and opened with a browser.

Troubleshooting

If you encounter any errors when building, please file an issue report at https://github.com/openscad/openscad/issues/ .

Errors about incompatible library versions

This may be caused by old libraries living in /usr/local/lib like boost, CGAL, OpenCSG, etc, (often left over from previous experiments with OpenSCAD). You are advised to remove them. To remove, for example, CGAL, run rm -rf /usr/local/include/CGAL && rm -rf /usr/local/lib/*CGAL*. Then erase $HOME/openscad_deps, remove your openscad source tree, and restart fresh. As of 2013 OpenSCAD's build process does not advise nor require anything to be installed in /usr/local/lib nor /usr/local/include.

Note that CGAL depends on Boost and OpenCSG depends on GLEW - interdependencies like this can really cause issues if there are stray libraries in unusual places.

Another source of confusion can come from running from within an 'unclean shell'. Make sure that you don't have LD_LIBRARY_PATH set to point to any old libraries in any strange places. Also don't mix a Mingw windows cross build with your linux build process - they use different environment variables and may conflict.

OpenCSG didn't automatically build

If for some reason the recommended build process above fails to work with OpenCSG, please file an issue on the OpenSCAD github. In the meantime, you can try building it yourself.

```
 wget http://www.opencsg.org/OpenCSG-1.3.2.tar.gz
 sudo apt-get purge libopencsg-dev libopencsg1 # or your system's equivalent
 tar -xvf OpenCSG-1.3.2.tar.gz
 cd OpenCSG-1.3.2
 # edit the Makefile and remove 'example'
 make
 sudo cp -d lib/lib* $HOME/openscad_deps/lib/
 sudo cp include/opencsg.h $HOME/openscad_deps/include/
```

Note: on Debian-based systems (such as Ubuntu), you can add the 'install' target to the OpenCSG Makefile, and then use checkinstall to create a clean .deb package for install/removal/upgrade. Add this target to Makefile:

```
install:
	# !! THESE LINES PREFIXED WITH ONE TAB, NOT SPACES !!
	cp -d lib/lib* /usr/local/lib/
	cp include/opencsg.h /usr/local/include/
	ldconfig
```

Then:

```
sudo checkinstall -D make install
```

.. to create and install a clean package.

CGAL didn't automatically build

If this happens, you can try to compile CGAL yourself. It is recommended to install to $HOME/openscad_deps and otherwise follow the build process as outlined above.

Compiling is horribly slow and/or grinds the disk

It is recommended to have at least 1.2 Gbyte of RAM to compile OpenSCAD. There are a few workarounds in case you don't. The first is to use the experimental support for the Clang Compiler (described below) as Clang uses much less RAM than GCC. Another workaround is to edit the Makefile generated by qmake and search/replace the optimization flags (-O2) with -O1 or blank, and to remove any '-g' debug flags from the compiler line, as well as '-pipe'.

If you have plenty of RAM and just want to speed up the build, you can try a paralell multicore build with

```
make -jx
```

Where 'x' is the number of cores you want to use. Remember you need x times the amount of RAM to avoid possible disk thrashing.

The reason the build is slow is because OpenSCAD uses template libraries like CGAL, Boost, and Eigen, which use large amounts of RAM to compile - especially CGAL. GCC may take up 1.5 Gigabytes of RAM on some systems during the build of certain CGAL modules. There is more information at StackOverflow.com (http://stackoverflow.com/questions/3634203/why-are-templates-so-slow-to-compile).

BSD issues

The build instructions above are designed to work unchanged on FreeBSD and NetBSD. However the BSDs typically require special environment variables set up to build any QT project - you can set them up automatically by running

```
 source ./scripts/setenv-unibuild.sh
```

NetBSD 5.x, requires a patched version of CGAL. It is recommended to upgrade to NetBSD 6 instead as it has all dependencies available from pkgin. NetBSD also requires the X Sets to be installed when the system was created (or added later (http://ghantoos.org/2009/05/12/my-first-shot-of-netbsd/)).

On OpenBSD it may fail to build after running out of RAM. OpenSCAD requires at least 1 Gigabyte to build with GCC. You may have need to be a user with 'staff' level access or otherwise alter required system parameters. The 'dependency build' sequence has also not been ported to OpenBSD so you will have to rely on the standard OpenBSD system package tools (in other words you have to have root).

Sun / Solaris / IllumOS / AIX / IRIX / Minix / etc

The OpenSCAD dependency builds have been mainly focused on Linux and BSD systems like Debian or FreeBSD. The 'helper scripts' likely will fail on other types of Un*x. Furthermore the OpenSCAD build system files (qmake .pro files for the GUI, cmake CMakeFiles.txt for the test suite) have not been tested thoroughly on non-Linux non-BSD systems. Extensive work may be required to get a working build on such systems.

Test suite problems

Headless server

The test suite will try to automatically detect if you have an X11 DISPLAY environment variable set. If not, it will try to automatically start Xvfb or Xvnc (virtual X framebuffers) if they are available.

If you want to run these servers manually, you can attempt the following:

```
$ Xvfb :5 -screen 0 800x600x24 &
$ DISPLAY=:5 ctest
```

Alternatively:

```
$ xvfb-run --server-args='-screen 0 800x600x24' ctest
```

There are some cases where Xvfb/Xvnc won't work. Some older versions of Xvfb may fail and crash without warning. Sometimes Xvfb/Xvnc have been built without GLX (OpenGL) support and OpenSCAD won't be able to generate any images.

Image-based tests takes a long time, they fail, and the log says 'return -11'

Imagemagick may have crashed while comparing the expected images to the test-run generated (actual) images. You can try using the alternate ImageMagick comparison method by by erasing CMakeCache, and re-running cmake with `-DCOMPARATOR=ncc`. This will enable the Normalized Cross Comparison method which is more stable, but possibly less accurate and may give false positives or negatives.

Testing images fails with 'morphology not found" for ImageMagick in the log

Your version of imagemagick is old. Upgrade imagemagick, or pass -DCOMPARATOR=old to cmake. The comparison will be of lowered reliability.

I moved the dependencies I built and now openscad won't run

It isn't advised to move them because the build is using RPATH hard coded into the openscad binary. You may try to workaround by setting the LD_LIBRARY_PATH environment variable to place yourpath/lib first in the list of paths it searches. If all else fails, you can re-run the entire dependency build process but export the BASEDIR environment variable to your desired location, before you run the script to set environment variables.

Tricks and tips

Reduce space of dependency build

After you have built the dependencies you can free up space by removing the $BASEDIR/src directory - where $BASEDIR defaults to $HOME/openscad_deps.

Preferences

OpenSCAD's config file is kept in ~/.config/OpenSCAD/OpenSCAD.conf.

Setup environment to start developing OpenSCAD in Ubuntu 11.04

The following paragraph describes an easy way to setup a development environment for OpenSCAD in Ubuntu 11.04. After executing the following steps QT Creator can be used to graphically start developing/debugging OpenSCAD.

- Add required PPA repositories:

```
# sudo add-apt-repository ppa:chrysn/openscad
```

- Update and install required packages:

```
# sudo apt-get update
# sudo apt-get install git build-essential qtcreator libglew1.5-dev libopencsg-dev libcgal-dev libeigen2-dev bison flex
```

- Get the OpenSCAD sources:

```
# mkdir ~/src
# cd ~/src
# git clone https://github.com/openscad/openscad.git
```

- Build OpenSCAD using the command line:

```
# cd ~/src/openscad
# qmake
# make
```

- Build OpenSCAD using QT Creator:

Just open the project file openscad.pro (CTRL+O) in QT Creator and hit the build all (CTRL+SHIFT+B) and run button (CTRL+R).

The Clang Compiler

There is experimental support for building with the Clang compiler under linux. Clang is faster, uses less RAM, and has different error messages than GCC. To use it, first of all you will need CGAL of at least version 4.0.2, as prior versions have a bug that makes clang unusable. Then, run this script before you build OpenSCAD.

```
source scripts/setenv-unibuild.sh clang
```

Clang support depends on your system's QT installation having a clang enabled qmake.conf file. For example, on Ubuntu, this is under /usr/share/qt4/mkspecs/unsupported/linux-clang/qmake.conf. BSD clang-building may require a good deal of fiddling and is untested, although eventually it is planned to move in this direction as the BSDs (not to mention OSX) are moving towards favoring clang as their main compiler. OpenSCAD includes convenience scripts to cross-build Windows installer binaries using the MXE system (http://mxe.cc). If you wish to use them, you can first install the MXE Requirements such as cmake, perl, scons, using your system's package manager (click to view a complete list of requirements) (http://mxe.cc/#requirements). Then you can perform the following commands to download OpenSCAD source and build a windows installer:

```
  git clone https://github.com/openscad/openscad.git
  cd openscad
  source ./scripts/setenv-mingw-xbuild.sh
  ./scripts/mingw-x-build-dependencies.sh
  ./scripts/release-common.sh mingw32
```

The x-build-dependencies process takes several hours, mostly to cross-build QT. It also requires several gigabytes of disk space. If you have multiple CPUs you can speed up things by running **export NUMCPU=x** before running the dependency build script. By default it builds the dependencies in $HOME/openscad_deps/mxe. You can override the mxe installation path by setting the BASEDIR environment variable before running the scripts. The OpenSCAD binaries are built into a separate build path, openscad/mingw32.

Note that if you want to then build linux binaries, you should log out of your shell, and log back in. The 'setenv' scripts, as of early 2013, required a 'clean' shell environment to work.

If you wish to cross-build manually, please follow the steps below and/or consult the release-common.sh source code.

Setup

The easiest way to cross-compile OpenSCAD for Windows on Linux or Mac is to use mxe (M cross environment). You will need to install git to get it. Once you have git, navigate to where you want to keep the mxe files in a terminal window and run:

```
git clone git://github.com/mxe/mxe.git
```

Add the following line to your `~/.bashrc` file:

```
export PATH=/<where mxe is installed>/usr/bin:$PATH
```

replacing `<where mxe is installed>` with the appropriate path.

Requirements

The requirements to cross-compile for Windows are just the requirements of mxe. They are listed, along with a command for installing them here (http://mxe.cc/#requirements). You don't need to type 'make'; this will make everything and take up >10 GB of diskspace. You can instead follow the next step to compile only what's needed for openscad.

Now that you have the requirements for mxe installed, you can build OpenSCAD's dependencies (CGAL, Opencsg, MPFR, and Eigen2). Just open a terminal window, navigate to your mxe installation and run:

```
make mpfr eigen opencsg cgal qt
```

This will take a few hours, because it has to build things like gcc, qt, and boost. Just go calibrate your printer or something while you wait. To speed things up, you might want do something like "make -j 4 JOBS=2" for parallel building. See the mxe tutorial (http://mxe.cc/#tutorial) for more details.

Optional: If you want to build an installer, you need to install the nullsoft installer system. It should be in your package manager, called "nsis".

Build OpenSCAD

Now that all the requirements have been met, all that remains is to build OpenSCAD itself. Open a terminal window and enter:

```
git clone git://github.com/openscad/openscad.git
cd openscad
```

Then get MCAD:

```
git submodule init
git submodule update
```

You need to create a symbolic link here for the build system to find the libraries:

```
ln -s /<where mxe is installed>/usr/i686-pc-mingw32/ mingw-cross-env
```

again replacing `<where mxe is installed>` with the appropriate path

Now to build OpenSCAD run:

```
i686-pc-mingw32-qmake CONFIG+=mingw-cross-env openscad.pro
make
```

When that is finished, you will have openscad.exe in ./release and you can build an installer with it as described in the instructions for building with Microsoft Visual C++, described here (http://en.wikibooks.org/wiki/OpenSCAD_User_Manual/Building_on_Windows#Building_an_installer).

The difference is that instead of right-clicking on the *.nsi file you will run:

```
makensis installer.nsis
```

Note that as of early 2013, OpenSCAD's 'scripts/release-common.sh' automatically uses the version of nsis that comes with the MXE cross build system, so you may wish to investigate the release-common.sh source code to see how it works, if you have troubles. This is a set of instructions for building OpenSCAD with the Microsoft Visual C++ compilers. It has not been used since circa 2012 and is unlikely to work properly. It is maintained here for historical reference purposes.

A newer build is being attempted with the Msys2 system, please see http://en.wikibooks.org/wiki/OpenSCAD_User_Manual/Building_on_Microsoft_Windows

---

This MSVC build is as static as reasonable, with no external DLL dependencies that are not shipped with Windows

Note: It was last tested on the Dec 2011 build. Newer checkouts of OpenSCAD may not build correctly or require extensive modification to compile under MSVC. OpenSCAD releases of 2012 were typically cross-compiled from linux using the Mingw & MXE system. See Cross-compiling for Windows on Linux or Mac OS X.

Downloads

start by downloading:

- Visual Studio Express http://download.microsoft.com/download/E/8/E/E8EEB394-7F42-4963-A2D8-29559B738298/VS2008ExpressWithSP1ENUX1504728.iso
- QT (for vs2008) http://get.qt.nokia.com/qt/source/qt-win-opensource-4.7.2-vs2008.exe
- git http://msysgit.googlecode.com/files/Git-1.7.4-preview20110204.exe
- glew https://sourceforge.net/projects/glew/files/glew/1.5.8/glew-1.5.8-win32.zip/download
- cmake http://www.cmake.org/files/v2.8/cmake-2.8.4-win32-x86.exe
- boost http://www.boostpro.com/download/boost_1_46_1_setup.exe
- cgal https://gforge.inria.fr/frs/download.php/27647/CGAL-3.7-Setup.exe
- OpenCSG http://www.opencsg.org/OpenCSG-1.3.2.tar.gz
- eigen2 http://bitbucket.org/eigen/eigen/get/2.0.15.zip
- gmp/mpfr http://holoborodko.com/pavel/downloads/win32_gmp_mpfr.zip
- MinGW http://netcologne.dl.sourceforge.net/project/mingw/Automated%20MinGW%20Installer/mingw-get-inst/mingw-get-inst-20110316/mingw-get-inst-20110316.exe

Installing

- Install Visual Studio
  - No need for siverlight or mssql express
  - You can use a virtual-CD program like MagicDisc to mount the ISO file and install without using a CD
- Install QT
  - Install to default location `C:\Qt\4.7.2\`
- Install Git
  - Click `Run Git and included Unix tools from the Windows Command Prompt` despite the big red letters warning you not to.
- Install Cmake
  - Check the 'Add cmake to the system path for the current user' checkbox
  - Install to default location `C:\Program Files\CMake 2.8`
- Install Boost

    - Select the VC++ 9.0 vs2008 radio
    - Check the 'multithreaded static runtime' checkbox only
    - Install into `C:\boost_1_46_1\`
- Install CGAL
    - Note - CGAL 3.9 fixes several bugs in earlier versions of CGAL, but CGAL 3.9 will not compile under MSVC without extensive patching. Please keep that in mind when compiling OpenSCAD with MSVC - there may be bugs due to the outdated version of CGAL required to use MSVC.
    - Note its not a binary distribution, just an installer that installs the source.
    - No need for CGAL Examples and Demos
    - Make sure mpfr and gmp precompiled libs is checked
    - The installer wants you to put this in `C:\Program Files\CGAL-3.7\` I used `C:\CGAL-3.7\`
    - Make sure CGAL_DIR environment checked.
- Install MinGW
    - Make sure you select the MSYS Basic System under components
- Extract downloaded win32_gmp_mpfr.zip file to `C:\win32_gmp_mpfr\`
- Replace the mpfr and gmp .h files in CGAL with the ones from win32_gmp_mpfr
    - Delete, or move to a temp folder, all files in `CGAL-3.7\auxiliary\gmp\include` folder
    - Copy all the .h files in `C:\win32_gmp_mpfr\gmp\Win32\Release` to `CGAL-3.7\auxiliary\gmp\include`
    - Copy all the .h files in `C:\win32_gmp_mpfr\mpfr\Win32\Release` to `CGAL-3.7\auxiliary\gmp\include`
- Replace the mpfr and gmp libs in CGAL with the ones from win32_gmp_mpfr
    - Delete, or move to a temp folder, all (06/20/2011 libmpfr-4.lib is needed 7/19/11 - i didnt need it) files in `CGAL-3.7\auxiliary\gmp\lib` folder.
    - Copy `C:\win32_gmp_mpfr\gmp\Win32\Release\gmp.lib` to `CGAL-3.7\auxiliary\gmp\lib`
    - Copy `C:\win32_gmp_mpfr\mpfr\Win32\Release\mpfr.lib` to `CGAL-3.7\auxiliary\gmp\lib`
    - Go into `CGAL-3.7\auxiliary\gmp\lib` and copy `gmp.lib` to `gmp-vc90-mt-s.lib`, and `mpfr.lib` to `mpfr-vc90-mt-s.lib` (so the linker can find them in the final link of openscad.exe)

To get OpenSCAD source code:

- Open "Git Bash" (or MingW Shell) (the installer may have put a shortcut on your desktop). This launches a command line window.
- Type **cd c:** to change the current directory.
- Type **git clone git://github.com/openscad/openscad.git** This will put OpenSCAD source into C:\openscad\

Where to put other files:

I put all the dependencies in C:\ so for example,

- C:\eigen2\
- C:\glew-1.5.8\
- C:\OpenCSG-1.3.2\

.tgz can be extracted with `tar -zxvf` from the MingW shell, or Windows tools like 7-zip. Rename and move sub-directories if needed. I.e eigen-eigen-0938af7840b0 should become c:\eigen2, with the files like COPYING and CMakeLists.txt directly under it. c:\glew-1.5.8 should have 'include' and 'lib' directly under it.

Compiling Dependencies

For compilation I use the QT Development Command Prompt

Start->Program Files->Qt by Nokia v4.7.2 (VS2008 OpenSource)->QT 4.7.2 Command Prompt

Qt

Qt needs to be recompiled to get a static C runtime build. To do so, open the command prompt and do:

```
configure -static -platform win32-msvc2008 -no-webkit
```

Configure will take several minutes to finish processing. After it is done, open up the file Qt\4.7.2\mkspecs\win32-msvc2008\qmake.conf and replace every instance of -MD with -MT. Then:

```
nmake
```

This takes a very, very long time. Have a nap. Get something to eat. On a Pentium 4, 2.8GHZ CPU with 1 Gigabyte RAM, Windows XP, it took more than 7 hours, (that was with -O2 turned off)

CGAL

```
cd C:\CGAL-3.7\
set BOOST_ROOT=C:\boost_1_46_1\
cmake .
```

Now edit the `CMakeCache.txt` file. Replace every instance of `/MD` with `/MT` . Now, look for a line like this:

`CMAKE_BUILD_TYPE:STRING=Debug`

Change `Debug` to `Release`. Now *re-run* cmake

```
cmake .
```

It should scroll by, watch for lines saying `"--Building static libraries"` and `"--Build type: Release"` to confirm the proper settings. Also look for `/MT` in the `CXXFLAGS` line. When it's done, you can do the build:

```
nmake
```

You should now have a `CGAL-vc90-mt-s.lib` file under `C:\CGAL-3.7\lib`. If not, see Troubleshooting, below.

OpenCSG

Launch Visual Express.

```
cd C:\OpenCSG-1.3.2
vcexpress OpenCSG.sln
Substitute devenv for vcexpress if you are not using the express version
```

- Manually step through project upgrade wizard
- Make sure the runtime library settings for all projects is for Release (not Debug)
  - Click Build/Configuration Manager
  - Select "Release" from "Configuration:" drop down menu
  - Hit Close
- Make sure the runtime library setting for OpenCSG project is set to multi-threaded static
  - Open the OpenCSG project properties by clicking menu item "Project->OpenCSG Properties" (might be just "Properties")
  - Make sure it says "Active(Release)" in the "Configuration:" drop down menu
  - Click 'Configuration Properties -> C/C++ -> Code Generation'
  - Make sure "Runtime Library" is set to "Multi-threaded (/MT)"
  - Click hit OK
- Make sure the runtime library setting for glew_static project is set to multi-threaded static
  - In "Solution Explorer - OpenCSG" pane click "glew_static" project
  - Open the OpenCSG project properties by clicking menu item "Project->OpenCSG Properties" (might be just "Properties")
  - Make sure it says "Active(Release)" in the "Configuration:" drop down menu
  - Click C/C++ -> Code Generation
  - Make sure "Runtime Library" is set to "Multi-threaded (/MT)"
  - Click hit OK
- Close Visual Express saving changes

Build OpenCSG library. You can use the GUI Build/Build menu (the Examples project might fail, but glew and OpenCSG should succeed). Alternatively you can use the command line:

```
cmd /c vcexpress OpenCSG.sln /build
Again, substitute devenv if you have the full visual studio
```

The `cmd /c` bit is needed otherwise you will be returned to the shell immediately and have to Wait for build process to complete (there will be no indication that this is happening appart from in task manager)

OpenSCAD

- Bison/Flex: Open the mingw shell and type `mingw-get install msys-bison`. Then do the same for flex: `mingw-get install msys-flex`

- Open the QT Shell, and copy/paste the following commands

```
cd C:\openscad
set INCLUDE=%INCLUDE%C:\CGAL-3.7\include;C:\CGAL-3.7\auxiliary\gmp\include;
set INCLUDE=%INCLUDE%C:\boost_1_46_1;C:\glew-1.5.8\include;C:\OpenCSG-1.3.2\include;C:\eigen2
set LIB=%LIB%C:\CGAL-3.7\lib;C:\CGAL-3.7\auxiliary\gmp\lib;
set LIB=%LIB%C:\boost_1_46_1\lib;C:\glew-1.5.8\lib;C:\OpenCSG-1.3.2\lib
qmake
nmake -f Makefile.Release
```

Wait for the nmake to end. There are usually a lot of non-fatal warnings about the linker. On success, there will be an openscad.exe file in the release folder. Enjoy.

Building an installer

- Download and install NSIS from http://nsis.sourceforge.net/Download
- Put the FileAssociation.nsh macro from http://nsis.sourceforge.net/File_Association in the NSIS Include directory, C:\Program Files\NSIS\Include
- Run 'git submodule init' and 'git submodule update' to download the MCAD system (https://github.com/elmom/MCAD) into the openscad/libraries folder.
- Copy the OpenSCAD "libraries" and "examples" directory into the "release" directory
- Copy OpenSCAD's "scripts/installer.nsi" to the "release" directory.
- Right-click on the file and compile it with NSIS. It will spit out a nice, easy installer. Enjoy.

Compiling the regression tests

- Follow all the above steps, build openscad, run it, and test that it basically works.
- Install Python 2.x (not 3.x) from http://www.python.org
- Install Imagemagick from http://www.imagemagick.org
- read openscad\docs\testing.txt
- Go into your QT shell

```
set PATH=%PATH%;C:\Python27 (or your version of python)
cd c:\openscad\tests\
cmake . -DCMAKE_BUILD_TYPE=Release
Edit the CMakeCache.txt file, search/replace /MD to /MT
cmake .
nmake -f Makefile
```

- This should produce a number of test .exe files in your directory. Now run

```
ctest
```

If you have link problems, see Troubleshooting, below.

Troubleshooting

Linker errors

If you have errors during linking, the first step is to improve debug logging, and redirect to a file. Open Openscad.pro and uncomment this line:

```
QMAKE_LFLAGS   += -VERBOSE
```

Now rerun

```
nmake -f Makefile.Release > log.txt
```

You can use a program like 'less' (search with '/') or wordpad to review the log.

To debug these errors, you must understand basics about Windows linking. Windows links to its standard C library with basic C functions like malloc(). But there are four different ways to do this, as follows:

```
compiler switch - type - linked runtime C library
/MT - Multithreaded static Release - link to LIBCMT.lib
/MTd - Multithreaded static Debug - link to LIBCMTD.lib
/MD - Multithreaded DLL Release - link to MSVCRT.lib (which itself helps link to the DLL)
/MDd - Multithreaded DLL Debug - link to MSVCRTD.lib (which itself helps link to the DLL)
```

All of the libraries that are link together in a final executable must be compiled with the same type of linking to the standard C library. Otherwise, you get link errors like, "LNK2005 - XXX is already defined in YYY". But how can you track down which library wasn't linked properly? 1. Look at the log, and 2. dumpbin.exe

dumpbin.exe

dumpbin.exe can help you determine what type of linking your .lib or .obj files were created with. For example, `dumpbin.exe /all CGAL.lib | find /i "DEFAULTLIB"` will give you a list of DEFAULTLIB symbols inside of CGAL.lib. Look for LIBCMT, LIBCMTD, MSVCRT, or MSVCRTD. That will tell you, according to the above table, whether it was built Static Release, Static Debug, DLL Release, or DLL Debug. (DLL, of course means Dynamic Link Library in this conversation.) This can help you track down, for example, linker errors about conflicting symbols in LIBCMT and LIBCMTD.

dumpbin.exe can also help you understand errors involving unresolved external symbols. For example, if you get an error about unresolved external symbol `___GLEW_NV_occlusion_query`, but your VERBOSE error log says the program linked in glew32.lib, then you can `dumpbin.exe /all glew32.lib | find /i "occlusion"` to see if the symbol is actually there. You may see a mangled symbol, with __impl, which gives you another clue with which you can google. In this particular example, glew32s.lib (s=static) should have been linked instead of glew32.lib.

CGAL

CGAL-vc90-mt-s.lib

After compilation, it is possible that you might get a file named `CGAL-vc90-mt.lib` or `CGAL-vc90-mt-gd.lib` instead of `CGAL-vc90-mt-s.lib`. There are many possibilities: you accidentally built the wrong version, or you may have built the right version and VCExpress named it wrong. To double check, and fix the problem, you can do the following:

```
cd C:\CGAL-3.7\lib
dumpbin /all CGAL-vc90-mt.lib | find /i "DEFAULTLIB"
(if you have mt-gd, use that name instead)
```

If this shows lines referencing `LIBCMTD, MSVCRT, or MSVCRTD` then you accidentally built the debug and/or dynamic version, and you need to clean the build, and try to build again with proper settings to get the *multi-threaded static release* version. However, if it just says `LIBCMT`, then you are probably OK. Look for another line saying `DEFAULTLIB:CGAL-vc90-mt-s`. If it is there, then you can probably just rename the file and have it work.

```
move CGAL-vc90-mt.lib CGAL-vc90-mt-s.lib
```

Visual Studio build

You can build CGAL using the GUI of visual studio, as an alternative to nmake. You have to use an alternate cmake syntax. Type 'cmake' by itself and it will give you a list of 'generators' that are valid for your machine; for example Visual Studio Express is `cmake -G"Visual Studio 9 2008" .`. That should get you a working `.sln` (solution) file.

Then run this:

```
vcexpress CGAL.sln
```

Modify the build configure target to Release (not Debug) and change the properties of the projects to be '/MT' multithreaded static builds. This is the similar procedure used to build OpenCSG, so refer to those instructions above for more detail.

Note for Unix users

The 'MingW Shell' (Start/Programs) provide tools like bash, sed, grep, vi, tar, &c. The C:\ drive is under '/c/'. MingW has packages, for example: `mingw-get install msys-unzip` downloads and installs the 'unzip' program. Git contains some programs by default, like perl. The windows command shell has cut/paste - hit `alt-space`. You can also change the scrollback buffer settings.

References

- Windows Building, OpenSCAD mailing list, 2011 May (http://rocklinux.net/pipermail/openscad/2011-May/thread.html).
- C Run-Time Libraries linking (http://msdn.microsoft.com/en-us/library/abx4dbyh(v=vs.80).aspx), Microsoft.com for Visual Studio 8 (The older manual is good too, here (http://msdn.microsoft.com/en-us/library/aa278396(VS.60).aspx))

- old nabble (http://old.nabble.com/flex-2.5.35-1:-isatty()-problem-(and-solution)-td17659695.html) on _isatty, flex
- Windows vs. Unix: Linking dynamic load modules (http://xenophilia.org/winvunix.html) by Chris Phoenix
- Static linking in CMAKE under MS Visual C (http://www.cmake.org/Wiki/CMake_FAQ#How_can_I_build_my_MSVC_application_with_a_static_runtime.3F) (cmake.org)
- __imp , declspec(dllimport), and unresolved references (http://stackoverflow.com/questions/3704374/linking-error-lnk2019-in-msvc-unresolved-symbols-with-imp-prefix-but-should) (stackoverflow.com)

For building OpenSCAD, see https://github.com/openscad/openscad/blob/master/README.md

For making release binaries, see http://svn.clifford.at/openscad/trunk/doc/checklist-macosx.txt

Libraries

Library Locations

OpenSCAD uses three library *locations*, the installation library, built-in library, and user defined libraries.

1. The *Installation* library location is the `libraries` directory under the directory where OpenSCAD is installed.
2. The *Built-In* library location is O/S dependent. Since version 2014.03, it can be opened in the system specific file manager using the "File->Show Library Folder..." menu entry.
   - Windows: `My Documents\OpenSCAD\libraries`
   - Linux: `$HOME/.local/share/OpenSCAD/libraries`
   - Mac OS X: `$HOME/Documents/OpenSCAD/libraries`
3. The *User-Defined* library path can be created using the `OPENSCADPATH` Environment Variable to point to the library(s). `OPENSCADPATH` can contain multiple directories in case you have library collections in more than one place, separate directories with a semi-colon for Windows, and a colon for Linux/Mac OS. For example:

   Windows: `C:\Users\A_user\Documents\OpenSCAD\MyLib;C:\Thingiverse Stuff\OpenSCAD Things;D:\test_stuff`
   (Note: For Windows, in versions prior to 2014.02.22 there is a bug preventing multiple directories in `OPENSCADPATH` as described above, it uses a colon (:) to separate directories. A workaround, if your libraries are on C: is to leave off the drive letter & colon, e.g. `\Thingiverse Stuff\OpenSCAD Things:\stuff`
   Linux/Mac OS: `/usr/lib:/home/mylib:.`

   OpenSCAD will need to be restarted to recognise any change to the `OPENSCADPATH` Environment Variable.

   Where you specify a *non-fully qualified* path & filename in the **`use <...>`** or **`include <...>`** statement that path/file is checked against the directory of the calling .scad file, the *User-Defined* library paths (`OPENSCADPATH`), the *Built-In* library (i.e. the O/S dependent locations above), and the *Installation* library, ***in that order***. NOTE: In the case of a library file itself having **`use <...>`** or **`include <...>`** the directory of the library .scad file is the 'calling' file, i.e. when looking for libraries within a library, it does not check the directory of the top level .scad file.

For example, with the following locations & files defined: (with `OPENSCADPATH=/usr/lib:/home/lib_os:.`)

```
1. <installation library>/lib1.scad
2. <built-in library>/lib2.scad
3. <built-in library>/sublib/lib2.scad
4. <built-in library>/sublib/lib3.scad
5. /usr/lib/lib2.scad
6. /home/lib_os/sublib/lib3.scad
```

The following **`include <...>`** statements will match to the nominated library files

```
include <lib1.scad>  // #1.
include <lib2.scad>  // #5.
include <sublib/lib2.scad>  // #3.
include <sublib/lib3.scad>  // #6.
```

Since 2014.03, the currently active list of locations can be verified in the "Help->Library Info" dialog.

The details info shows both the content of the `OPENSCADPATH` variable and the list of all library locations. The locations will be searched in the order they appear in this list. For example;

```
OPENSCADPATH: /data/lib1:/data/lib2
OpenSCAD library path:
  /data/lib1
  /data/lib2
  /home/user/.local/share/OpenSCAD/libraries
  /opt/OpenSCAD/libraries
```

Setting `OPENSCADPATH`

In Windows, Environment Variables are set via the `Control panel`, select `System`, then `Advanced System Settings`, click `Environment Variables`. Create a new `User Variable`, or edit `OPENSCADPATH` if it exists.

On Linux (probably also on Mac), to simply add the environment variable to all users, you can type in terminal: `sudo sh -c 'echo "OPENSCADPATH=$HOME/openscad/libraries" >>/etc/profile'` to set the `OPENSCADPATH` to `openscad/libraries` under each user's home directory. For more control on environment variables, you'll need to edit the configuration files; see for example this page (http://unix.stackexchange.com/questions/117467/how-to-permanently-set-environmental-variables).

MCAD

OpenSCAD bundles the MCAD library (https://github.com/openscad/MCAD).

There are many different forks floating around (e.g.[2] (https://github.com/SolidCode/MCAD), [3] (https://github.com/elmom/MCAD), [4] (https://github.com/benhowes/MCAD)) many of them unmaintained.

MCAD bundles a lot of stuff, of varying quality, including:

- Many common shapes like rounded boxes, regular polygons and polyeders in 2D and 3D
- Gear generator for involute gears and bevel gears.
- Stepper motor mount helpers, stepper and servo outlines
- Nuts, bolts and bearings
- Screws and augers
- Material definitions for common materials
- Mathematical constants, curves
- Teardrop holes and polyholes

The git repo also contains python code to scrape OpenSCAD code, a testing framework and SolidPython, an external python library for solid cad.

Other Libraries

- BOLTS tries to build a standard part and vitamin library that can be used with OpenSCAD and other CAD tools: [5] (http://www.bolts-library.org/)
- Obiscad contains various useful tools, notably a framework for attaching modules on other modules in a simple and modular way: [6] (https://github.com/Obijuan/obiscad)
- This library provides tools to create proper 2D technical drawings of your 3D objects: [7] (http://www.cannymachines.com/entries/9/openscad_dimensioned_drawings)
- Stephanie Shaltes (https://plus.google.com/u/0/101448691399929440302) wrote a fairly comprehensive fillet library (https://github.com/StephS/i2_xends/blob/master/inc/fillets.scad)
- The shapes library (http://svn.clifford.at/openscad/trunk/libraries/shapes.scad) contains many shapes like rounded boxes, regular polygons. It is also included in MCAD.
- Also Giles Bathgates shapes library (https://github.com/elmom/MCAD/blob/master/regular_shapes.scad) provides regular polygons and polyeders and is included in MCAD.
- The OpenSCAD threads (http://dkprojects.net/openscad-threads/) library provides ISO conform metric and imperial threads and support internal and external threads and multiple starts.
- Sprockets for ANSI chains and motorcycle chains can be created with the Roller Chain Sprockets OpenSCAD Module (http://www.thingiverse.com/thing:197896). Contains hard coded fudge factors, may require tweaking.
- The Pinball Library (http://code.google.com/p/how-to-build-a-pinball/source/browse/trunk/scad/pinball) provides many components for pinball design work, including models for 3d printing of the parts, 3d descriptions of mount holes for CNC drilling and 2d descriptions of parts footprint
- For the generation of celtic knots there is the Celtic knot library (https://github.com/beanz/celtic-knot-scad)
- The 2D connection library (https://www.youmagine.com/designs/openscad-2d-connection-library) helps with connections between 2D sheets, which is useful for laser cut designs.
- local.scad (https://github.com/jreinhardt/local-scad) provides a flexible method for positioning parts of a design. Is also used in BOLTS.
- SCADBoard (http://scadboard.wordpress.com/) is a library for designing 3D printed PCBs in OpenSCAD.
- A Ruler (http://www.thingiverse.com/thing:30769) for determining the size of things in OpenSCAD.
- A colorspace converter for working with colors in HSV and RGB: http://www.thingiverse.com/thing:279951/
- A number of utility functions is collected in https://github.com/oampo/missile
- Unit test framework https://github.com/oampo/testcard
- Knurled surface library by aubenc http://www.thingiverse.com/thing:9095
- Text module based on technical lettering style https://github.com/thestumbler/alpha

There is also a list with more libraries here: [8] (https://github.com/openscad/openscad/wiki/Libraries)

Command Glossary

This is a Quick Reference; a short summary of all the commands without examples, just the basic syntax. The headings are links to the full chapters.

Please be warned: The Command Glossary is presently outdated (03 2015).

Please have a look at the Cheatsheet, instead:

http://www.openscad.org/cheatsheet/

Mathematical Operators

```
+
-   // also as unary negative
*
/
%
```

```
<
<=
==
!=
>=
>
```

```
&&   // logical and
||   // logical or
!    // logical not
```

```
<boolean> ? <valIfTrue> : <valIfFalse>
```

Mathematical Functions

```
abs ( <value> )
```

```
cos ( <degrees> )
sin ( <degrees> )
tan ( <degrees> )
asin ( <value> )
acos ( <value> )
atan ( <value> )
atan2 ( <y_value>, <x_value> )
```

```
pow( <base>, <exponent> )
```

```
len ( <string> )   len ( <vector> )   len ( <vector_of_vectors> )
min ( <value1>, <value2> )
max ( <value1>, <value2> )
sqrt ( <value> )
round ( <value> )
ceil ( <value> )
floor ( <value> )
lookup( <in_value>, <vector_of_vectors> )
```

String Functions

```
str(string, value, ...)
```

Primitive Solids

```
cube(size = <value or vector>, center = <boolean>);
```

```
sphere(r = <radius>);
```

```
cylinder(h = <height>, r1 = <bottomRadius>, r2 = <topRadius>, center = <boolean>);
cylinder(h = <height>, r = <radius>);
```

```
polyhedron(points = [[x, y, z], ... ], triangles = [[p1, p2, p3..], ... ], convexity = N);
```

Transformations

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

```
(In versions > 2013.03)
resize(newsize=[x,y,z], auto=(true|false) { ... }
resize(newsize=[x,y,z], auto=[xaxis,yaxis,zaxis]) { ... }  // #axis is true|false
resize([x,y,z],[xaxis,yaxis,zaxis]) { ... }
resize([x,y,z]) { ... }
```

```
rotate(a = deg, v = [x, y, z]) { ... }
rotate(a=[x_deg,y_deg,z_deg]) { ... }
```

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

```
mirror([ 0, 1, 0 ]) { ... }
```

```
multmatrix(m = [tranformationMatrix]) { ... }
```

```
color([r, g, b, a]) { ... }
color([ R/255, G/255, B/255, a]) { ... }
color("blue",a) { ... }
```

Conditional and Iterator Functions

```
for (<loop_variable_name> = <vector> ) {...}
```

```
intersection_for (<loop_variable_name> = <vector_of_vectors>) {...}
```

```
if (<boolean condition>) {...} else {...}
```

```
assign (<var1>= <val1>, <var2>= <val2>, ...) {...}
```

CSG Modelling

```
union() {...}
```

```
difference() {...}
```

```
intersection() {...}
```

```
render(convexity = <value>) { ... }
```

Modifier Characters

```
! { ... } // Ignore the rest of the design and use this subtree as design root
* { ... } // Ignore this subtree
% { ... } // Ignore CSG of this subtree and draw it in transparent gray
# { ... } // Use this subtree as usual but draw it in transparent pink
```

Modules

```
module name(<var1>, <var2>, ...) { ...<module code>...}
```

Variables can be default initialized `<var1>=<defaultvalue>`

In module you can use `children()` to refer to all child nodes, or `children(i)` where `i` is between `0` and `$children`.

Include Statement

After 2010.02

```
include <filename.scad> (appends whole file)
```

```
use <filename.scad>  (appends ONLY modules and functions)
```

filename could use directory (with / char separator).

Prior to 2010.02

```
<filename.scad>
```

Other Language Features

```
$fa is the minimum angle for a fragment. The default value is 12 (degrees)
```

```
$fs is the minimum size of a fragment. The default value is 1.
```

```
$fn is the number of fragments. The default value is 0.
```

When $fa and $fs are used to determine the number of fragments for a circle, then OpenSCAD will never use less than 5 fragments.

```
$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.

```
function name(<var>) = f(<var>);
```

```
echo(<string>, <var>, ...);
```

```
render(convexity = <val>) {...}
```

```
surface(file = "filename.dat", center = <boolean>, convexity = <val>);
```

2D Primitives

```
square(size = <val>, center=<boolean>);
square(size = [x,y], center=<boolean>);
```

```
circle(r = <val>);
```

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

3D to 2D Projection

```
projection(cut = <boolean>)
```

2D to 3D Extrusion

```
linear_extrude(height = <val>, center = <boolean>, convexity = <val>, twist = <degrees>[, slices = <val>, $fn=...,$fs=...,$fa=...]){...}
```

```
rotate_extrude(convexity = <val>[, $fn = ...]){...}
```

DXF Extrusion

```
linear_extrude(height = <val>, center = <boolean>, convexity = <val>, twist = <degrees>[...])
import (file = "filename.dxf", layer = "layername")
```

```
rotate_extrude(origin = [x,y], convexity = <val>[, $fn = ...])
import (file = "filename.dxf", layer = "layername")
```

STL Import

```
import("filename.stl", convexity = <val>);
```