OpenSCAD Help

Table of contents

Introduction

OpenSCADHelp

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

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

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

The libraries included in this project are the next:

MCAD

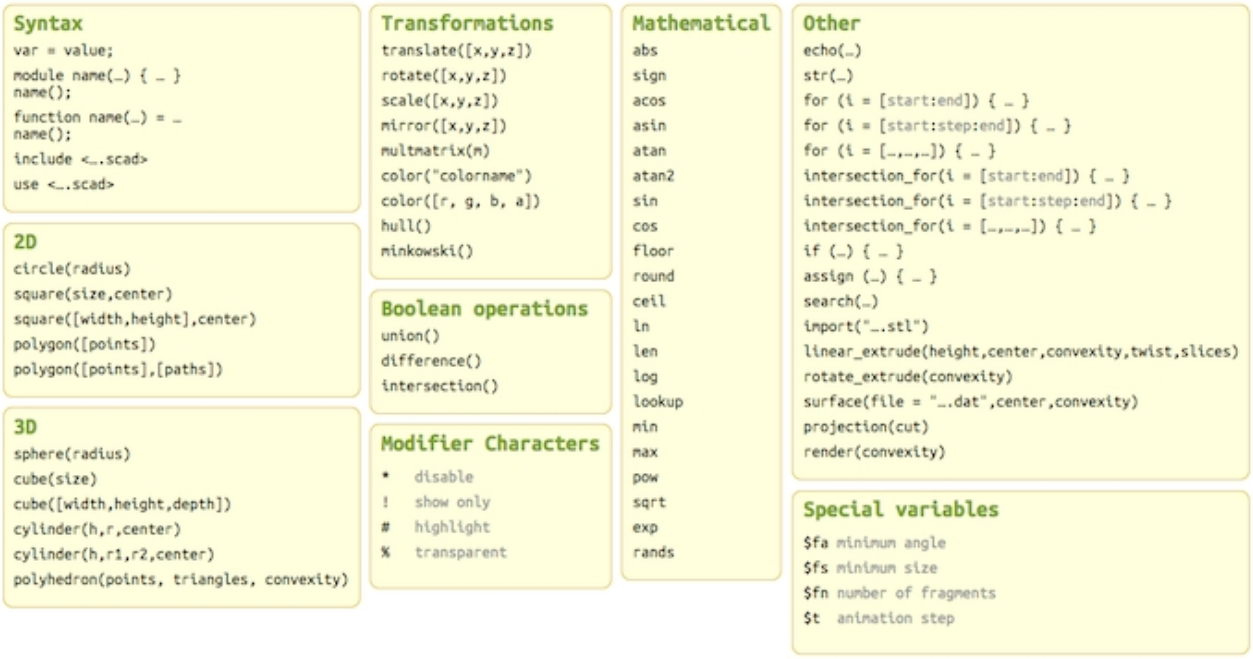
Components commonly used in designing and mocking up mechanical designs https://github.com/openscad/MCAD

dotSCAD

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

BOSL

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



What's new

2020 April 22th.

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

Getting Started

OpenSCAD is a software for creating solid 3D CAD objects.
It is free software and available for GNU/Linux, Microsoft Windows and Mac OS X.

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

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

OpenSCAD has two main operating modes:

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

OpenSCAD provides two types of 3D modelling:

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

OpenSCAD can be downloaded from https://www.openscad.org/. **OpenSCAD** is a software for creating solid 3D CAD objects.
It is free software and available for GNU/Linux, Microsoft Windows and Mac OS X.

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

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

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

OpenSCAD provides two types of 3D modelling:

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

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

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

Language Reference

Language Reference

Table of Contents

Introduction

Introduction

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

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

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

Objects

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

Actions

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

Operators

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

Examples

```
cube(5);
x = 4 + y;
rotate(40) square(5,10);
translate([10,5]) { circle(5); square(4); }
rotate(60) color("red") { circle(5); square(4); }
```

color("blue") { translate([5,3,0]) sphere(5); rotate([45,0,45]) { cylinder(10); cube([5,6,7]); } }

Comments

Comments

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

OpenSCAD uses C++-style comments:

```
// This is a comment

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

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


Values and Data Types

Values and Data Types

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

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

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

Numbers

Numbers

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 a 64 bit IEEE floating point number.

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

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

- We use binary floating point. A fractional number is not represented exactly unless the denominator is a power of 2. For example, 0.2 (2/10) does not have an exact internal representation, but 0.25 (1/4) and 0.125 (1/8) are represented exactly.
- The largest representable number is about 1e308. If a numeric result is too large, then the result can be infinity (printed as inf by echo).
- The smallest representable number is about -1e308. If a numeric result is too small, then the result can be -infinity (printed as -inf by echo).
- If a numeric result is invalid, then the result can be Not A Number (printed as **nan** by echo).
- If a non-zero numeric result is too close to zero to be representable, then the result is -0 if the result is negative, otherwise it is 0. Zero (0) and negative zero (-0) are treated as two distinct numbers by some of the math operations, and are printed differently by 'echo', although they compare as equal.

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

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

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

Boolean Values

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

- `false`
- `0 and -0`
- `""`

- `[]`
- `undef`

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

Strings

Strings

A string is a sequence of zero or more unicode characters. String values are used to specify file names when importing a file, and to display text for debugging purposes when using echo(). Strings can also be used with the text() primitive.
A string literal is written as a sequence of characters enclosed in quotation marks `"`, like this: `""` (an empty string), or `"this is a string"`.
To include a `"` character in a string literal, use `\"`. To include a `\` character in a string literal, use `\\`. The following escape sequences beginning with `\` can be used within string literals:

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

Ranges

Ranges

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

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

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

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

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

A missing <*increment*> defaults to 1.

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

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

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

The Undefined Value

The Undefined Value

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

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

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

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

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

Variables

Variables

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

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

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

which are typically contrasted with normal variables.

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

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

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

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

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


Undefined variable

Undefined variable

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

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


Scope of variables

Scope of variables

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

Each pair of braces creates a new scope inside the scope where they were used. New variables

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

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

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

Anonymous scopes are not considered scopes:

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

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

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

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

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

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

While this appears to be counter-intuitive, it allows you to do some interesting things: for instance, if you set up your shared library files to have default values defined as variables at their

root level, when you include that file in your own code, you can 're-define' or override those constants by simply assigning a new value to them.

Special Variables

Special Variables

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

Vectors

Vectors

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

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

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

examples

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

use in OpenSCAD:

```
cube( [width, depth, height]);      // optional spaces shown for clarity
translate( [x, y, z] )
polygon( [ [x0, y0], [x1, y1], [x2, y2] ] );
```

creation

Vectors are created by writing the list of elements, separated by commas, and enclosed in

square brackets. Variables are replaced by their values.

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

elements within vectors

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

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

example elements with lengths from len()

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

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

alternate dot notation

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

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


Vector Operators (concat & len)

Vector Operators (concat & len)

concat

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

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

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

len

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

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

string
- Returns the number of characters in string.

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


Matrix

Matrix

A matrix is a vector of vectors.

Example which defines a 2D rotation matrix

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


Getting input

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

```
OriginPoint = 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.

3D Objects

Primitive Solids

cube

cube

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

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

parameters:

size

single value, cube with all sides this length

3 value array [x,y,z], cube with dimensions x, y and z.

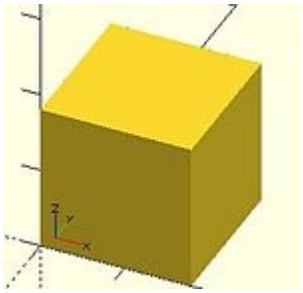
center

false (default), 1st (positive) octant, one corner at (0,0,0)

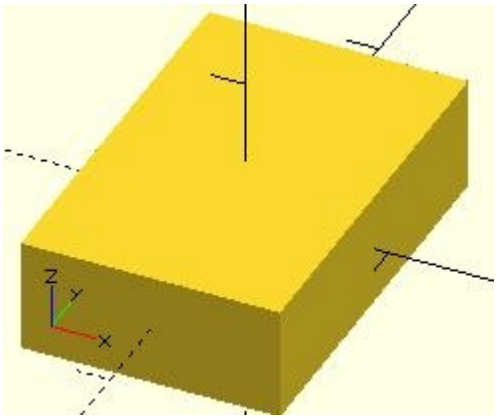
true, cube is centered at (0,0,0)

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

examples:



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



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


sphere

sphere

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

Parameters

r
Radius. This is the radius of the sphere. The resolution of the sphere is based on the size of the sphere and the $fa, $fs and $fn variables. For more information on these special variables look at: OpenSCAD_User_Manual/Other_Language_Features
d

Diameter. This is the diameter of the sphere.
$fa
Fragment angle in degrees
$fs
Fragment size in mm
$fn
Resolution

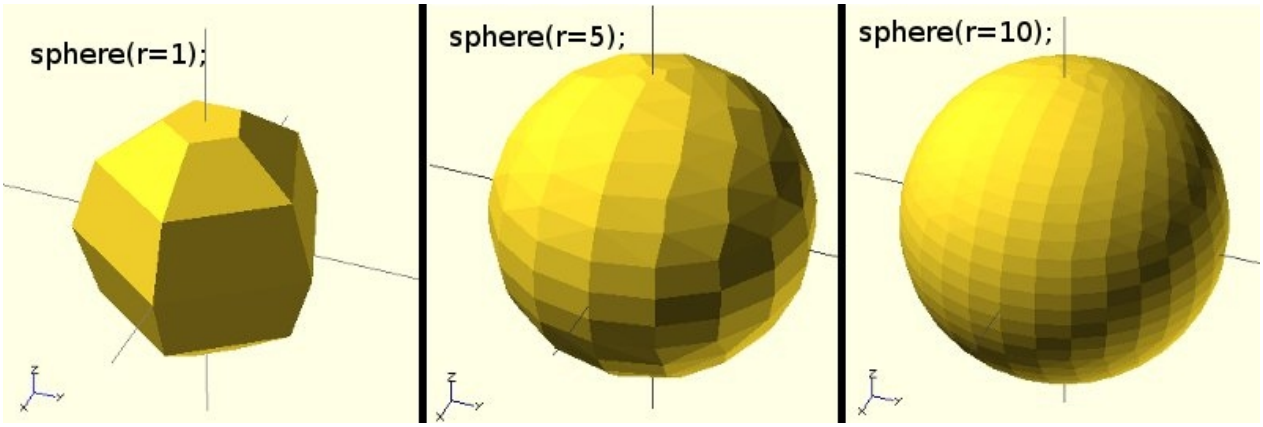
```
default values: sphere(); yields: sphere($fn = 0, $fa = 12, $fs =
2, r = 1);
```

Usage Examples

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

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

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



cylinder

cylinder

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

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

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

```
cylinder(h = height, r1 = BottomRadius, r2 = TopRadius, center =
```

```
true/false);
```

Parameters

h : height of the cylinder or cone

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

r1 : radius, bottom of cone.

r2 : radius, top of cone.

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

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

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

center

false (default), z ranges from 0 to h

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

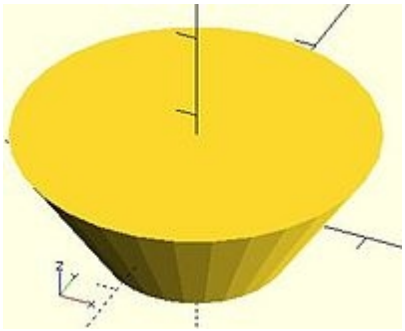
$fa : minimum angle (in degrees) of each fragment.

$fs : minimum circumferential length of each fragment.

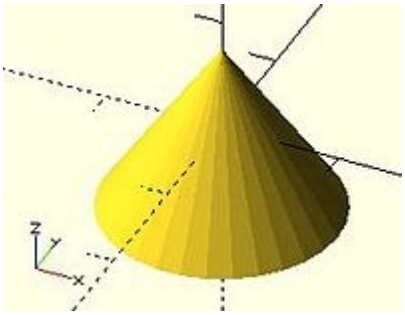
$fn : **fixed** number of fragments in 360 degrees. Values of 3 or more override $fa and $fs

$fa, $fs and $fn must be named. click here for more details,.

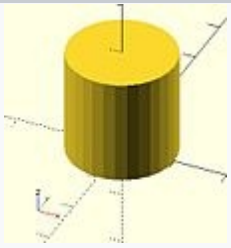
```
defaults: cylinder(); yields: cylinder($fn = 0, $fa = 12, $fs = 2,
h = 1, r1 = 1, r2 = 1, center = false);
```



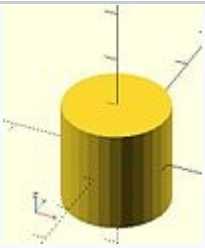
```
equivalent scripts cylinder(h=15, r1=9.5, r2=19.5, center=false);
cylinder( 15, 9.5, 19.5, false); cylinder( 15, 9.5, 19.5);
cylinder( 15, 9.5, d2=39 ); cylinder( 15, d1=19, d2=39 );
cylinder( 15, d1=19, r2=19.5);
```



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



center = false



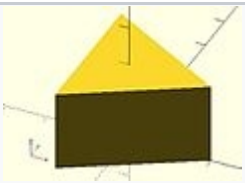
center = true

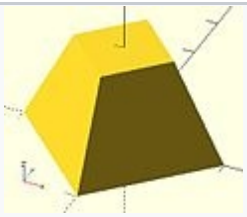
```
equivalent scripts cylinder(h=20, r=10, center=true);
cylinder( 20, 10, 10,true); cylinder( 20, d=20, center=true);
cylinder( 20,r1=10, d2=20, center=true); cylinder( 20,r1=10,
d2=2*10, center=true);
```

use of $fn

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

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

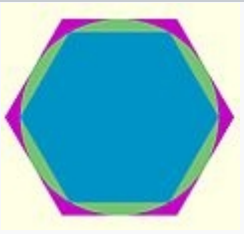


- 

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

undersized holes

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

- 

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

polyhedron

polyhedron

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

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

Parameters

points

Vector of 3d points or vertices. Each point is in turn a vector, [x,y,z], of its coordinates.

Points may be defined in any order. N points are referenced, in the order defined, as 0 to N-1.

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

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

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

Vector of faces that collectively enclose the solid. Each face is a vector containing the indices (0 based) of 3 or more points from the points vector.
Faces may be defined in any order. Define enough faces to fully enclose the solid, with no overlap.
If points that describe a single face are not on the same plane, the face is automatically split into triangles as needed.

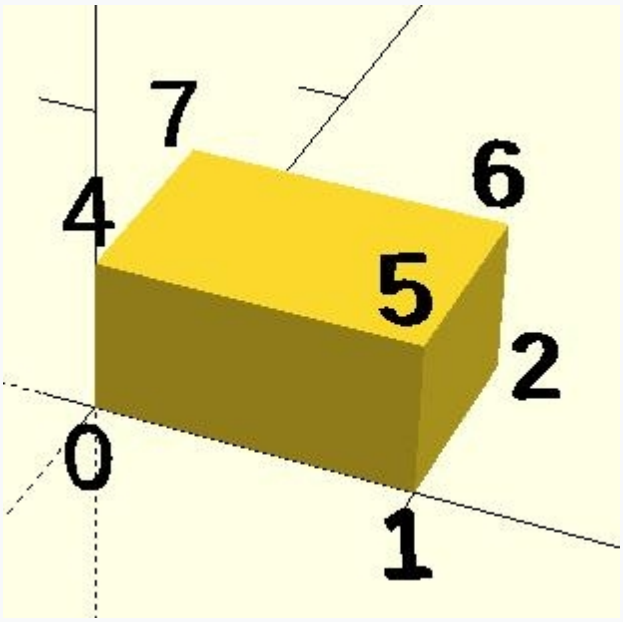
convexity

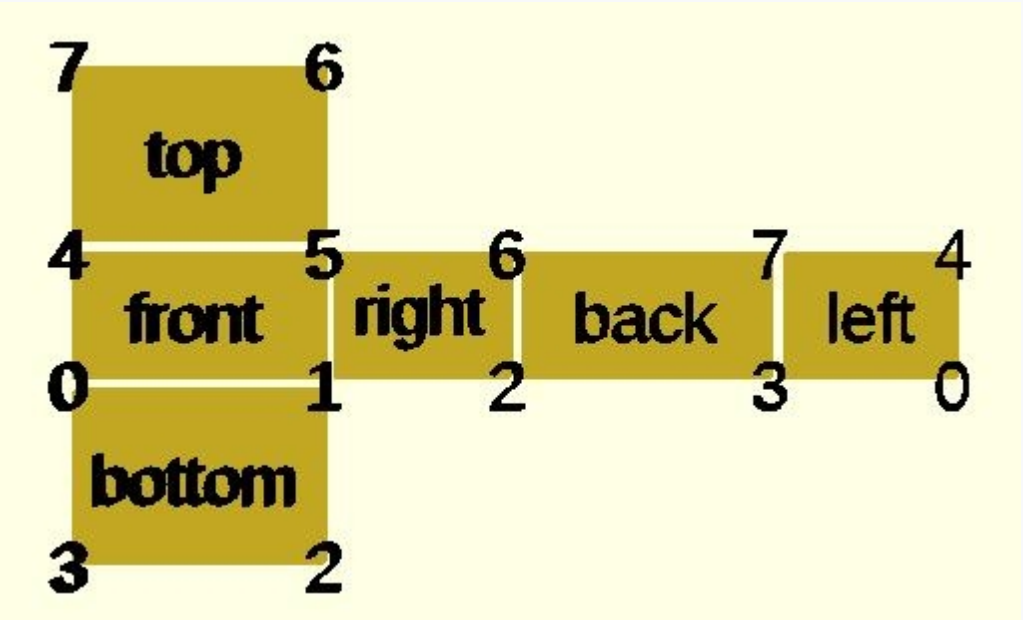
Integer. The convexity parameter specifies the maximum number of faces a ray intersecting the object might penetrate. This parameter is needed only for correct display of the object in OpenCSG preview mode. It has no effect on the polyhedron rendering. For display problems, setting it to 10 should work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

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

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

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



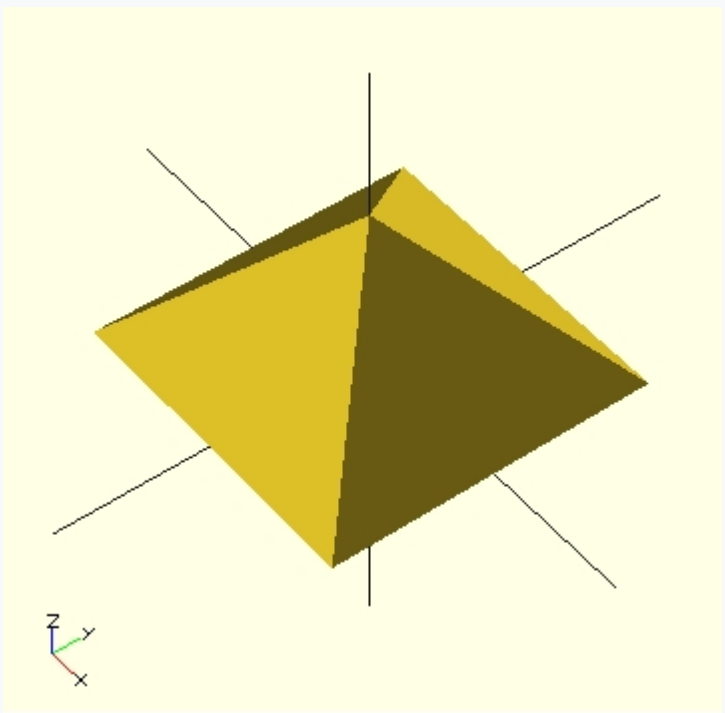


unfolded cube faces

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

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

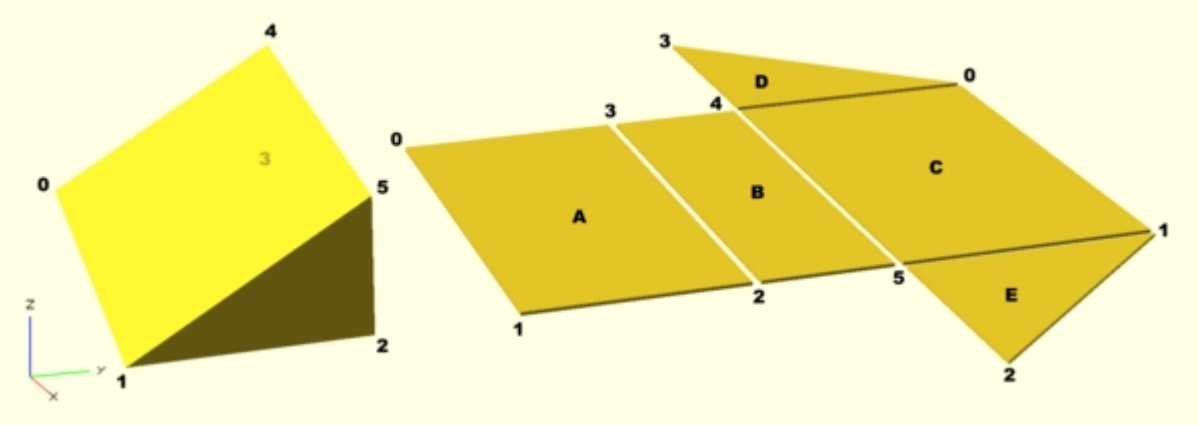
Example 2 A square base pyramid:



A simple polyhedron, square base pyramid

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

Example 3 A triangular prism:



A polyhedron triangular prism

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

point numbers for cube

Debugging polyhedra

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

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

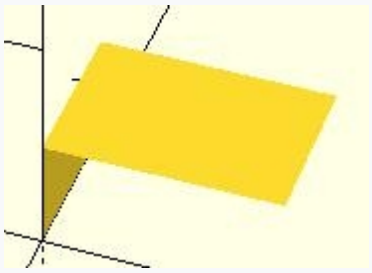
The first rule eliminates polyhedra like two cubes with a common edge and not watertight models; the second excludes polyhedra like two cubes with a common vertex.

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

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

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

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



example 1 showing only 2 faces

Mis-ordered faces

Mis-ordered faces

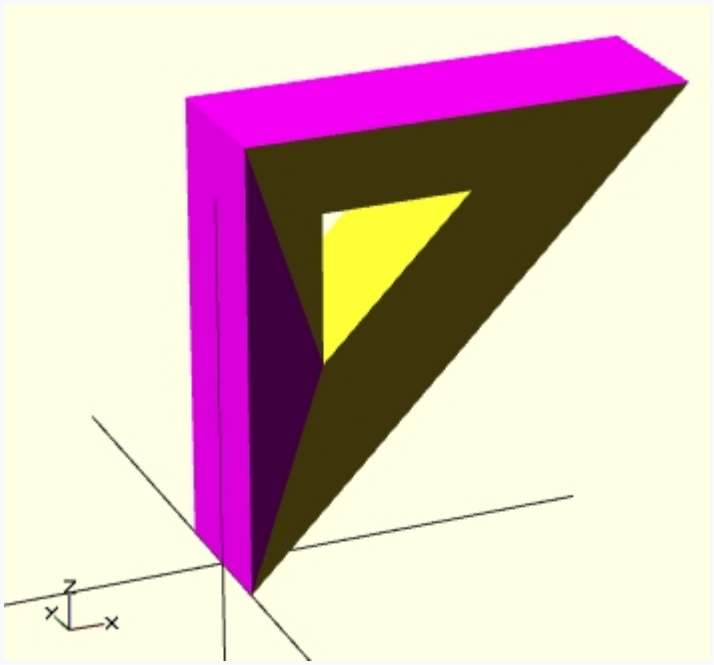
Example 4 a more complex polyhedron with mis-ordered faces

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

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

```
// Bad polyhedron
polyhedron
(points = [
```

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



Polyhedron with badly oriented polygons

A correct polyhedron would be the following:

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


Point repetitions in a polyhedron point list

Point repetitions in a polyhedron point list

Beginner's tip

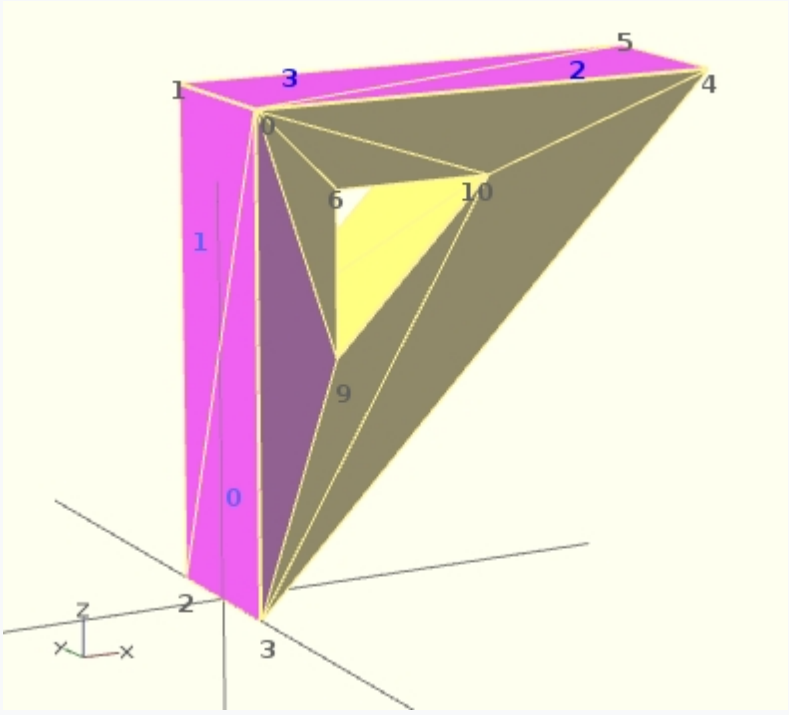
If you don't really understand "orientation", try to identify the mis-oriented pink faces and then invert the sequence of the references to the points vectors until you get it right. E.g. in the above example, the third triangle (*[0,4,5]*) was wrong and we fixed it as *[4,0,5]*. Remember that a face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

Clockwise Technique

Orientation is determined by clockwise circular indexing. This means that if you're looking at the triangle (in this case [4,0,5]) from the outside you'll see that the path is clockwise around the center of the face. The winding order [4,0,5] is clockwise and therefore good. The winding order [0,4,5] is counter-clockwise and therefore bad. Likewise, any other clockwise order of [4,0,5] works: [5,4,0] & [0,5,4] are good too. If you use the clockwise technique, you'll always have your faces outside (outside of OpenSCAD, other programs do use counter-clockwise as the outside though).

Think of it as a Left Hand Rule:

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



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 enumerated starting from zero for use in the faces list (0,1,2,3,... etc).

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

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

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

Point repetitions in a polyhedron point list[edit]

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

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

define the same tetrahedron as:

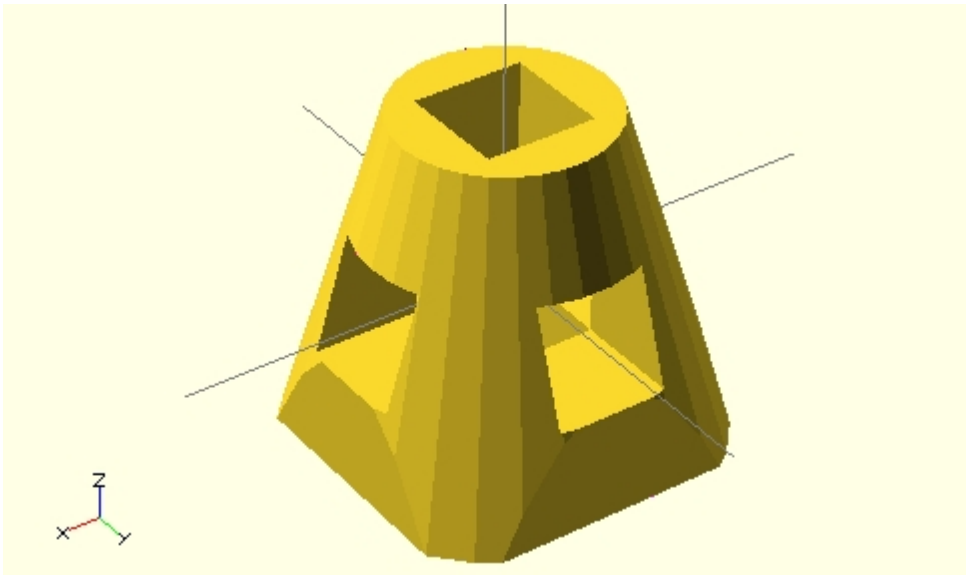
```
points = [[0,0,0], [0,10,0], [10,0,0], [0,0,10]];
polyhedron(points, [[0,2,1], [0,1,3], [1,2,3], [0,3,2]]);
```


3D to 2D Projection

3D to 2D Projection

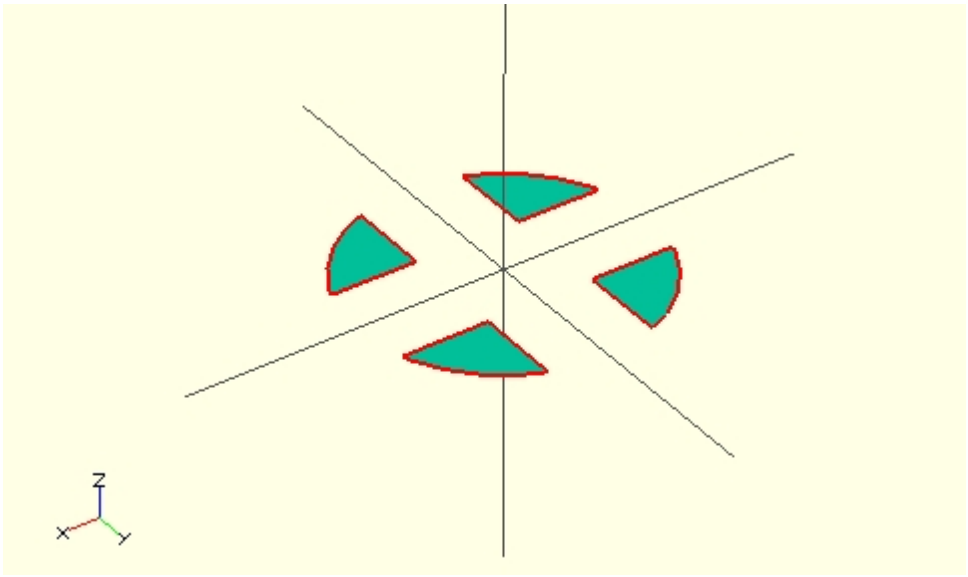
Using the `projection()` function, you can create 2d drawings from 3d models, and export them to the dxf format. It works by projecting a 3D model to the (x,y) plane, with z at 0. If `cut=true`, only points with z=0 are considered (effectively cutting the object), with `cut=false`(*the default*), points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

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



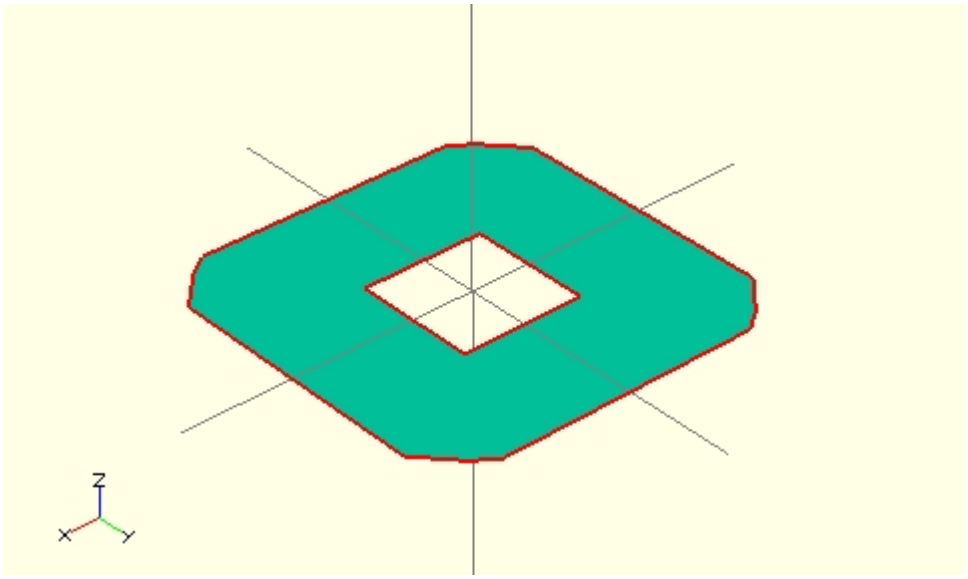
Then you can do a 'cut' projection, which gives you the 'slice' of the x-y plane with z=0.

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



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

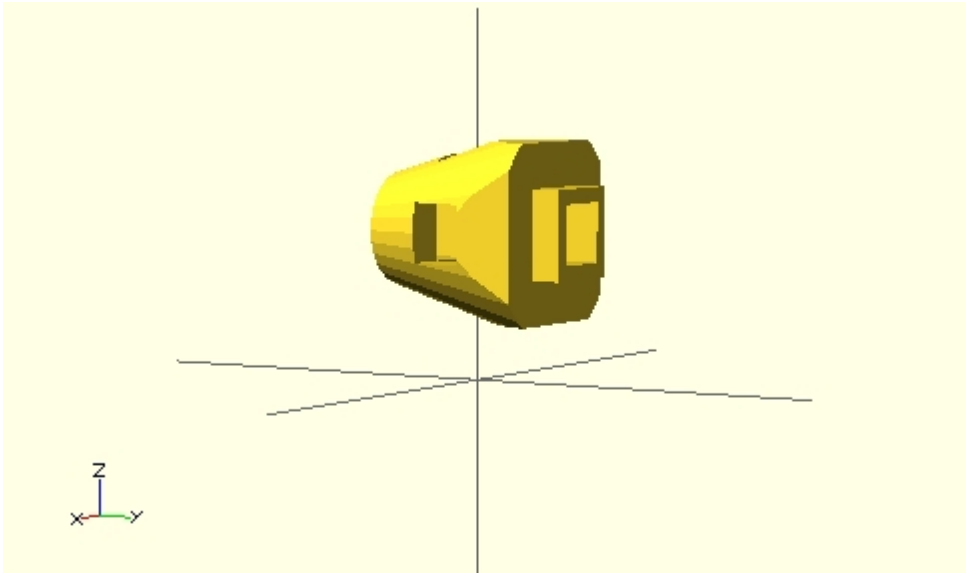
```
projection(cut = false) example002();
```



Another Example

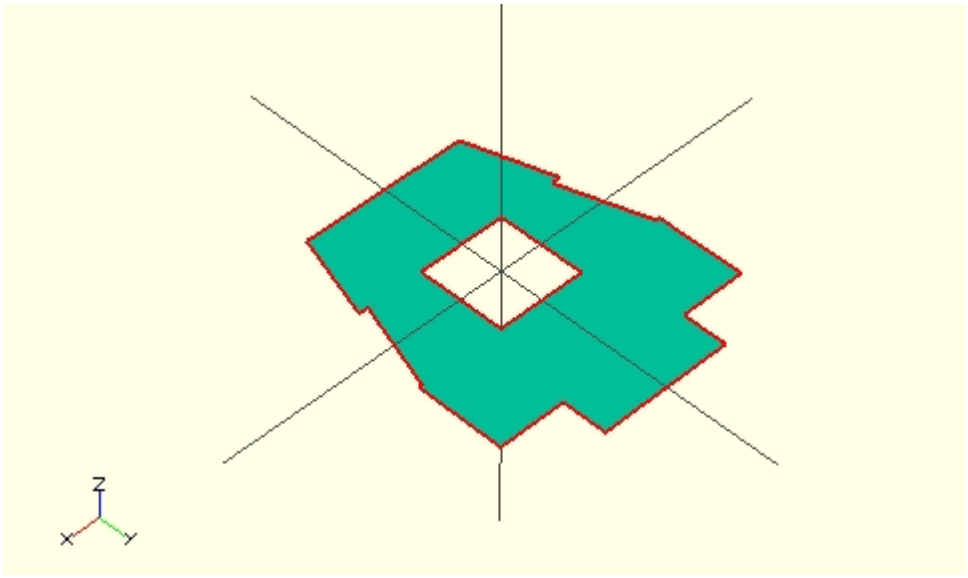
You can also use projection to get a 'side view' of an object. Let's take example002, and move it up, out of the X-Y plane, and rotate it:

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



Now we can get a side view with projection()

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



Links:

- example021.scad from Clifford Wolf's site.
- More complicated example from Giles Bathgate's blog

2D Objects

2D Objects

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

square

square

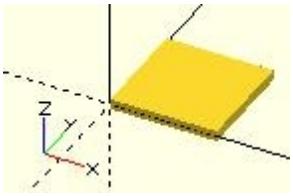
Creates a square or rectangle in the first quadrant. When center is true the square is centered on the origin. Argument names are optional if given in the order shown here.

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

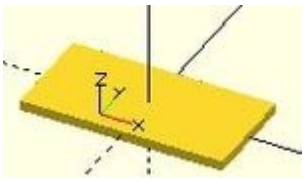
parameters:
size
single value, square with both sides this length
2 value array [x,y], rectangle with dimensions x and y
center
false (default), 1st (positive) quadrant, one corner at (0,0)
true, square is centered at (0,0)

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

examples:



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



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


circle

circle

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

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

Parameters

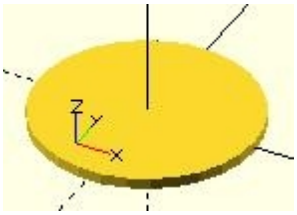
r : circle radius. r name is the only one optional with circle.
circle resolution is based on size, using $fa or $fs.

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

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

d : circle diameter (only available in versions later than 2014.03).
$fa : minimum angle (in degrees) of each fragment.
$fs : minimum circumferential length of each fragment.
$fn : **fixed** number of fragments in 360 degrees. Values of 3 or more override $fa and $fs
$fa, $fs and $fn must be named. click here for more details,.

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

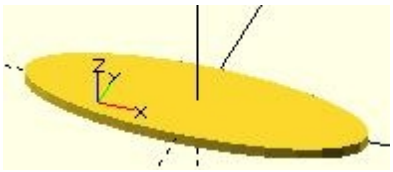


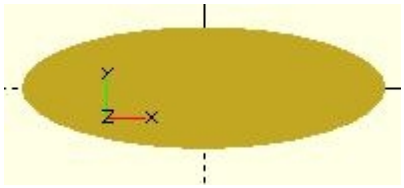
```
equivalent scripts for this example circle(10); circle(r=10); circle(d=20); circle(d=2+9*2);
```


ellipse

ellipse

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





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


regular polygon

regular polygon

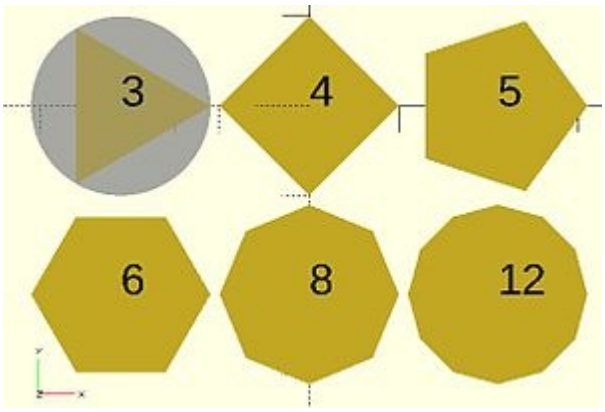
A regular polygon of 3 or more sides can be created by using circle() with $fn set to the number of sides. The following two pieces of code are equivalent.

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

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

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

the polygon primitive below.




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

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


polygon

polygon

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

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

Parameters

points

The list of x,y points of the polygon. : A vector of 2 element vectors.
Note: points are indexed from 0 to n-1.

paths

default

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

single vector

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

multiple vectors

Creates primary and secondary shapes. Secondary shapes are subtracted from the primary shape (like difference). Secondary shapes may be wholly or partially within the primary shape.

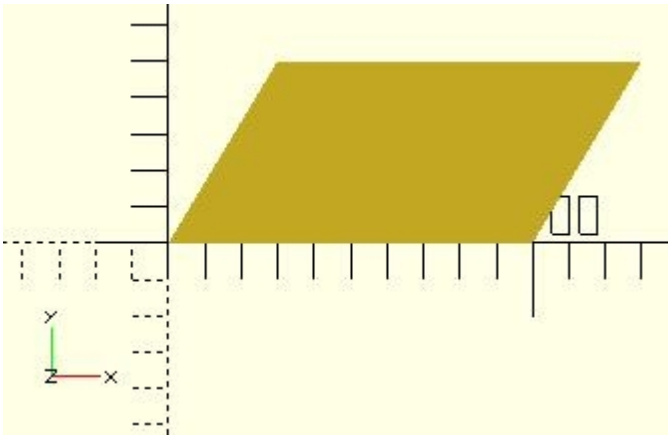
A closed shape is created by returning from the last point specified to the first.

convexity

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

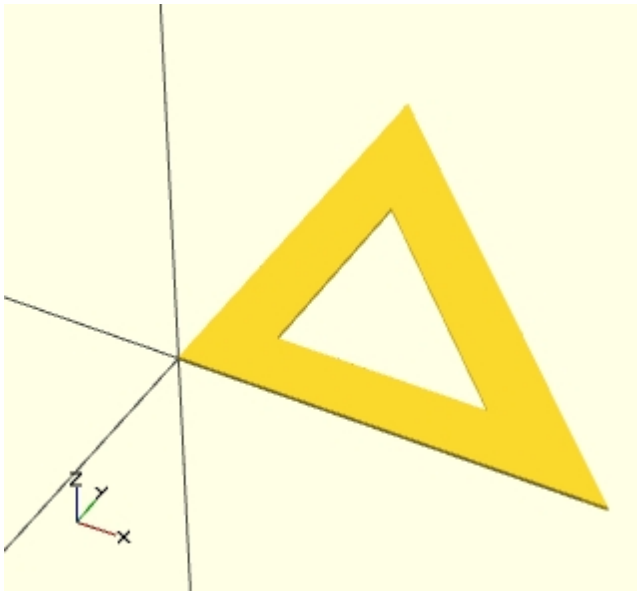
```
defaults: polygon(); yields: polygon(points = undef, paths =
undef, convexity = 1);
```

Example no holes



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

Example one hole

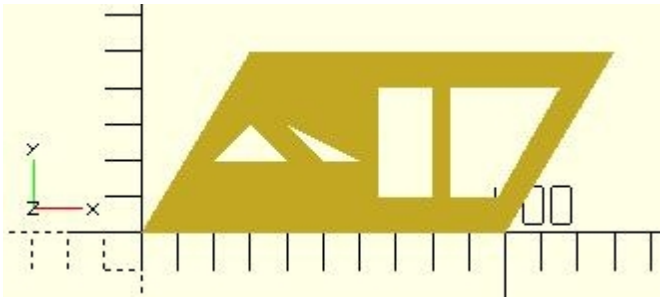


```
equivalent scripts for this example polygon(points=[[0,0],[100,0],
[0,100],[10,10],[80,10],[10,80]], paths=[[0,1,2],
[3,4,5]],convexity=10); triangle_points =[[0,0],[100,0],[0,100],
[10,10],[80,10],[10,80]]; triangle_paths =[[0,1,2],[3,4,5]];
polygon(triangle_points,triangle_paths,10);
```

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

Example multi hole

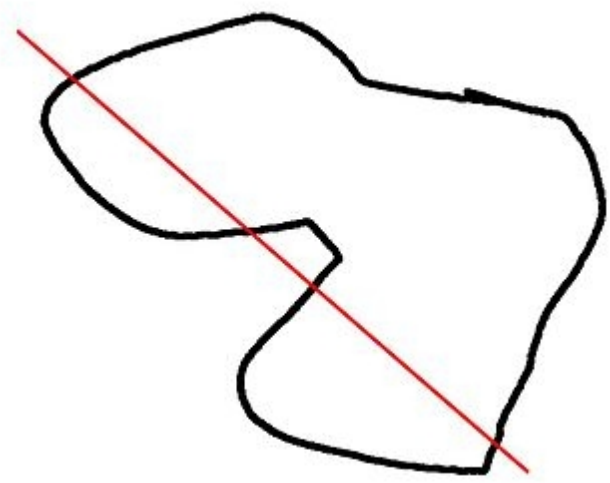
[Note: *Requires version **2015.03***] (for use of `concat()`)




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

convexity

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



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

Text

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

Parameters

text
String. The text to generate.
size
Decimal. The generated text has an ascent (height above the baseline) of approximately the given value. Default is 10. Different fonts can vary somewhat and may not fill the size specified exactly, typically they render slightly smaller.
font
String. The name of the font that should be used. This is not the name of the font file, but the logical font name (internally handled by the fontconfig library). This can also include a style parameter, see below. A list of installed fonts & styles can be obtained using the font list dialog (Help -> Font List).
halign
String. The horizontal alignment for the text. Possible values are "left", "center" and "right". Default is "left".
valign
String. The vertical alignment for the text. Possible values are "top", "center", "baseline" and "bottom". Default is "baseline".
spacing
Decimal. Factor to increase/decrease the character spacing. The default value of 1 results in the normal spacing for the font, giving a value greater than 1 causes the letters to be spaced further

apart.
direction
String. Direction of the text flow. Possible values are "ltr" (left-to-right), "rtl" (right-to-left), "ttb" (top-to-bottom) and "btt" (bottom-to-top). Default is "ltr".
language
String. The language of the text. Default is "en".
script
String. The script of the text. Default is "latin".
$fn
used for subdividing the curved path segments provided by freetype

Example

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

Note

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

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

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

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

Example

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

Example 1: Result.

Using Fonts & Styles

Alignment

Vertical Alignment

Horizontal Aligment

3D Text

3D to 2D Projection

2D to 3D Projection

Linear Extrude

Usage

Twist

Center

Mesh Refinement

Scale

Rotate Extrude

Parameters

Usage

Examples

Mesh Refinement

Extruding a Polygon

Description of extrude Parameters

Extrude parameters for all extrusion modes

Extrude parameters for linear extrusion only

Transforms

Basic concept

Advanced concept

scale

resize

rotate

Rotation rule Help

translate

mirror

Function signature:

Examples

multimatrix

More?

color

function signature

Example

Example 2

offset

minkowski

hull

Combining transformations

Boolean Combinations

Boolean overview

2D examples

3D examples

union

difference

difference with multiple children

intersection

render

Other Functions and Operators

Condition and Iterator functions

For loop

Intersection For loop

if statement

else if

Conditional ?

Recursive function calls

Assign Statement

Let Statement

Mathematical Operators

Scalar Arithmetical Operators

Relational Operators

Logical Operators

Conditional Operator

Vector-Number Operator

Vector Operators

Vector Dot-Product Operator

Matrix Multiplication

Mathematical Functions

Trigonometric Functions

cos

sin

tan

acos

asin

atan

atan2

Other Mathematical Functions

abs

ceil

concat

cross

exp

floor

in

len

let

log

lookup

max

min

norm

pow

rands

round

sign

sqrt

Infinites and NaNs

String Functions

str

chr

ord

Also see search()

List Comprehensions

Basic Syntax

Multiple generator expressions

for

each

if

if/else

let

Nested loops

Advanced Examples

Generating vertices for a polygon

Flattening a nested vector

Sorting a vector

Selecting elements of a vector

Concatenating two vectors

Other Language Features

Special Variables

$fa, $fs and $fn

$t

$vpr, $vpt and $vpd

$preview

Echo Statements

Usage examples

Rounding examples

Small and large Numbers

HTML

Echo Function

Render

Surface

Text file format

Images

Examples

Search

Search Usage

Search Arguments

Search Usage Examples

Index values return as list

Search on different column; return index values

Search on list of values

Search on list of strings

Getting the right result

OpenSCAD Version

parent_module(n) and $parent_modules

assert

Example

Checking parameters

Adding message

Using assertions in functions

User Defined Functions and Modules

Introduction

Scope

Functions

Recursive Functions

Function Literals

Modules

Object modules

Operator Modules

Children

Further Module Examples

Recursive Modules

Overwriting bult-in modules

Overwriting built functions

Debugging Aids

Advanced Concept

Background Modifier

Debug Modifier

Root Modifier

Disable Modifier

Echo Statements

External Libraries and code files

Use and Include

Directory separators

Variables

Scope of Variables

Overwriting variables

Example "Ring-Library"

Nested Include and Use

Import

Parameters

Convexity

Notes

Import DXF

Import STL

surface

Parameters

Text file Format

Images

Examples

MCAD Library

MCAD Library

This library contains components commonly used in designing and mocking up mechanical designs. It is currently unfinished and you can expect some API changes, however many things are already working.

This library was created by various authors as named in the individual files' comments. All the files are

licensed under the LGPL 2.1 (see http://creativecommons.org/licenses/LGPL/2.1/ or the included file lgpl-2.1.txt), some of them allow distribution under more permissive terms (as described in the files' comments).

Usage

You can import these files in your scripts with use <MCAD/filename.scad>, where 'filename' is one of the files listed below like 'motors' or 'servos'. Some files include useful constants which will be available with include <MCAD/filename.scad>, which should be safe to use on all included files (ie. no top level code should create geometry).

(There is a bug/feature that prevents including constants from files that "include" other files - see the openscad mailing list archives for more details. Since the maintainers aren't very responsive, may have to work around this somehow)

If you host your project in git, you can do git submodule add URL PATH in your repo to import this library as a git submodule for easy usage. Then you need to do a git submodule update --init after cloning. When you want to update the submodule, do cd PATH; git checkout master; git pull. See git help submodule for more info.

Currently Provided Tools:

regular_shapes.scad
    regular polygons, ie. 2D
    regular polyhedrons, ie. 3D
involute_gears.scad (http://www.thingiverse.com/thing:3575):
    gear()
    bevel_gear()
    bevel_gear_pair()
gears.scad (Old version):
    gear(number_of_teeth, circular_pitch OR diametrial_pitch, pressure_angle OPTIONAL, clearance OPTIONAL)
motors.scad:
    stepper_motor_mount(nema_standard, slide_distance OPTIONAL, mochup OPTIONAL)

Tools (alpha and beta quality):

nuts_and_bolts.scad: for creating metric and imperial bolt/nut holes
bearing.scad: standard/custom bearings
screw.scad: screws and augers
materials.scad: color definitions for different materials
stepper.scad: NEMA standard stepper outlines
servos.scad: servo outlines
boxes.scad: box with rounded corners
triangles.scad: simple triangles
3d_triangle.scad: more advanced triangles

Very generally useful functions and constants:

math.scad: general math functions
constants.scad: mathematical constants
curves.scad: mathematical functions defining curves
units.scad: easy metric units
utilities.scad: geometric funtions and misc. useful stuff
teardrop.scad (http://www.thingiverse.com/thing:3457): parametric teardrop module
shapes.scad: DEPRECATED simple shapes by Catarina Mota
polyholes.scad: holes that should come out well when printed

Other:

alphabet_block.scad
bitmap.scad
letter_necklace.scad
name_tag.scad
height_map.scad
trochoids.scad
libtriangles.scad
layouts.scad
transformations.scad
2Dshapes.scad
gridbeam.scad
fonts.scad
unregular_shapes.scad
metric_fastners.scad
lego_compatibility.scad
multiply.scad
hardware.scad

External utils that generate and process openscad code:

openscad_testing.py: testing code, see below
openscad_utils.py: code for scraping function names etc.

regular_shapes.scad

2D regular shapes

regular_polygon(sides, radius)

n-gons 2D shapes

triangle(radius)

pentagon(radius)

hexagon(radius)

heptagon(radius)

octagon(radius)

nonagon(radius)

decagon(radius)

hendecagon(radius)

dodecagon(radius)

ring(inside_diameter, thickness)

ellipse(width, height)

egg_outline(width, thickness)

3D regular shapes

cone

oval_prism

oval_tube

cylinder_tube

triangle_prism

triangle_tube

pentagon_prism

pentagon_tube

hexagon_prism

hexagon_tube

heptagon_prism

heptagon_tube

octagon_prism

octagon_tube

nonagon_prism

decagon_prism

hendecagon_prism

dodecagon_prism

torus

torus2

oval_torus

triangle_pyramid

square_pyramid

egg

involute_gears.scad

bevel_gear_pair()

bevel_gear()

gear()

Tests

test_gears()

test_meshing_double_helix()

test_bevel_gear()

test_bevel_gear_pair()

test_backlash()

dotSCAD Library

2D modules

arc

pie

rounded_square

line2d

polyline2d

hull_polyline2d

hexagons

polytransversals

multi_line_text

voronoi2d

3D modules

rounded_cube

rounded_cylinder

crystal_ball

line3d

polyline3d

hull_polyline3d

function_grapher

sweep

loft

starburst

voronoi3d

Transformations

along_width

hollow_out

bend

shear

2D functions

in_shape

bijection_offset

trim_shape

triangulate

contours

2D/3D functions

cross_sections

paths2sections

path_scaling_sections

bezier_surface

bezier_smooth

midpt_smooth

in_polyline

Paths

arc_path

bspline_curve

bezier_curve

helix

golden_spiral

archimedean_spiral

sphere_spiral

torus_knot

Extrusion

box_extrude

ellipse_extrude

stereographic_extrude

rounded_extrude

bend_extrude

2D Shape

shape_taiwan

shape_arc

shape_pie

shape_circle

shape_ellipse

shape_square

shape_trapezium

shape_cyclicpolygon

shape_pentagram

shape_starbusrst

shape_superformula

shape_glue2circles

shape_path_extend

2D Shape extrusions

path_extrude

ring_extrude

helix_extrude

golden_spiral_extrude

archimedean_spiral_extrude

sphere_spiral_extrude

Utils

util/sub_str

util/split_str

util/parse_number

util/reverse

util/slice

util/sort

util/rand

util/fibseq

util/bsearch

util/has

util/dedup

util/flat

Matrix

matrix/m_cumulate

matrix/m_translation

matrix/m_rotation

matrix/m_scaling

matrix/m_mirror

matrix/m_shearing

Point Transformation

ptf/ptf_rotate

ptf/ptf_x_twist

ptf/ptf_y_twist

ptf/ptf_circle

ptf/ptf_bend

ptf/ptf_ring

ptf/ptf_sphere

ptf/ptf_torus

Turtle

turtle/turtle2d

turtle/turtle3d

turtle/t2d

turtle/t3d

Pixel

pixel/px_line

pixel/px_polyline

pixel/px_circle

pixel/px_cylinder

pixel/px_sphere

pixel/px_polygon

pixel/px_from

pixel/px_ascii

pixel/px_gray

Part

part/connector_jpg

part/cone

part/join_T

Surface

surface/sf_square

surface/sf_bend

surface/sf_ring

surface/sf_sphere

surface/sf_torus

surface/sf_solidity

Noise

noise/nz_perlin1

noise/nz_perlin1s

noise/nz_perlin2

noise/nz_perlin2s

noise/nz_perlin3

noise/nz_perlin3s

noise/nz_worley2

noise/nz_worley2s

noise/nz_worley3

noise/nz_worley3s

noise/nz_cell

BOSL Library

Commonly Used

Commonly Used

The most commonly used transformations, manipulations, and shortcuts.

transforms.scad

Translations

move()

move()

use<transforms.scad>

Usage:

- move([x], [y], [z]) ...

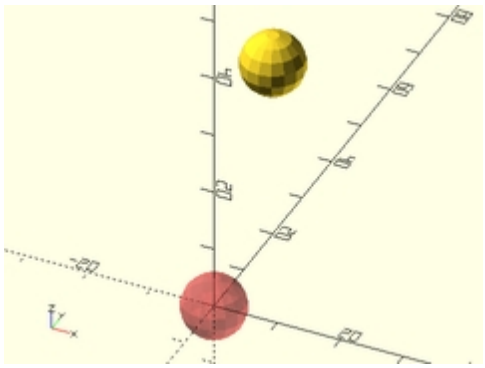
- move([x, y, z]) ...

Description: Moves/translates children.

Argument	What it does
x	X axis translation.
y	Y axis translation.
z	Z axis translation.

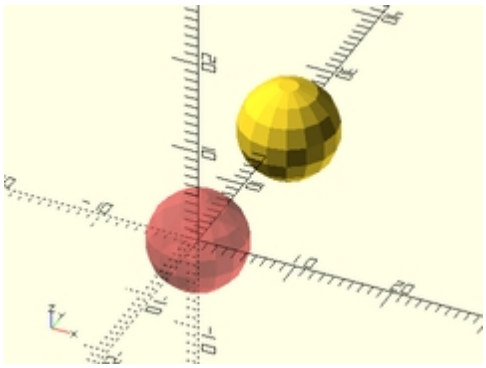
Example 1:

```
#sphere(d = 10);
move([0, 20, 30]) sphere(d = 10);
```



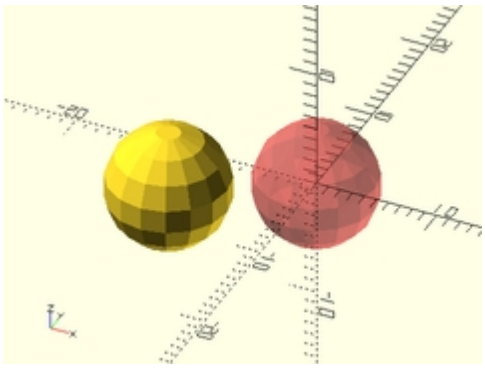
Example 2:

```
#sphere(d = 10);
move(y = 20) sphere(d = 10);
```



Example 3:

```
#sphere(d = 10);
move(x = -10, y = -5) sphere(d = 10);
```



xmove()

xmove()

use<transforms.scad>

Usage:

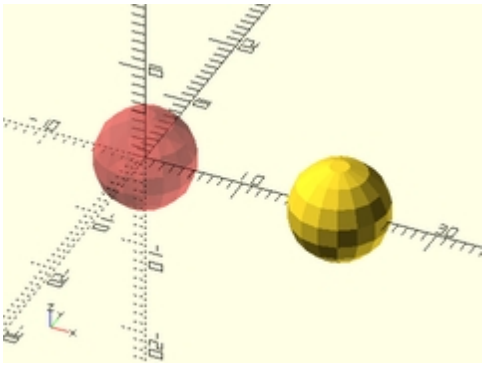
- xmove(x) ...

Description: Moves/translates children the given amount along the X axis.

Argument	What it does
x	Amount to move right along the X axis. Negative values move left.

Example:

```
#sphere(d = 10);
xmove(20) sphere(d = 10);
```



ymove()

zmove()

left()

right()

fwd() / fordware()

back()

down()

up()

shapes.scad

masks.scad

threading.scad

paths.scad

beziers.scad

Standard Parts

involute_gears.scad

joiners.scad

sliders.scad

metric_screws.scad

linear_bearings.scad

nema_steppers.scad

phillips_drive.scad

torx_drive.scad

wiring_scad

Miscelaneous

constants.scad

math.scad

convex_hull.scad

quaternions.scad

triangulation.scad

debug.scad

Nut Job Library