User Guide

Z3 Development edited this page 19 days ago · 285 revisions

OpenJSCAD.org Version 0.019 (2015/01/07) - Note: This Is Work In Progress

Introduction

Welcome to **OpenJSCAD.org User & Programming Guide**.

Just for sake of providing context, **OpenJSCAD.org** is built on OpenJsCad (Github), which itself was inspired by OpenSCAD.org, and essentially provides a programmers approach with **JavaScript** to develop 3D models, in particular this enhancement is tuned toward creating models for 3D printing.

OpenJSCAD is heavily object-oriented, and for programmers coming from **OpenSCAD** may welcome

- support for OpenSCAD source-code (apprx. 95% functions available) and
- there are a few JavaScript functions which ease the transition to OpenJSCAD as well,

Pages 11

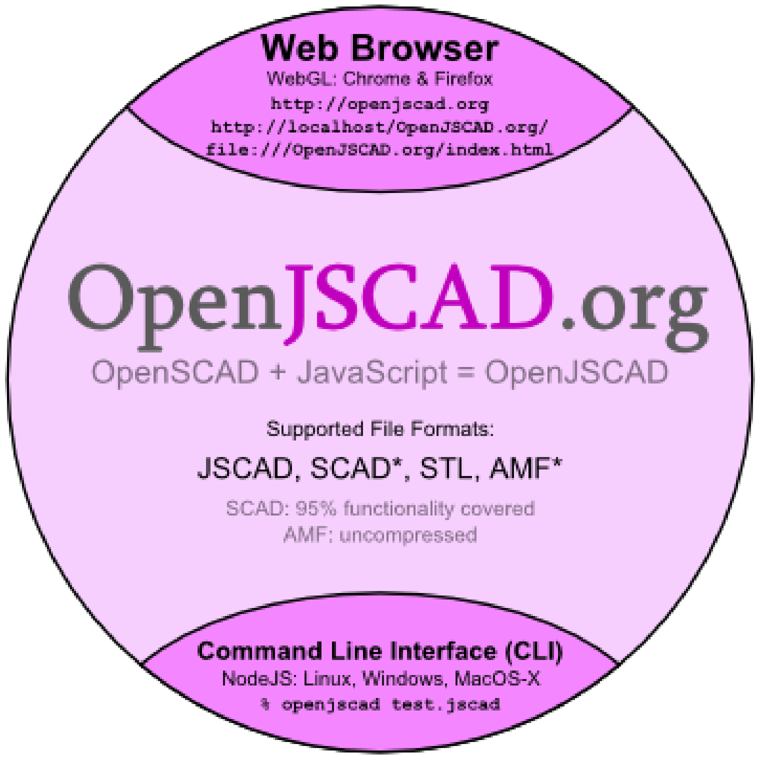
Table of Contents

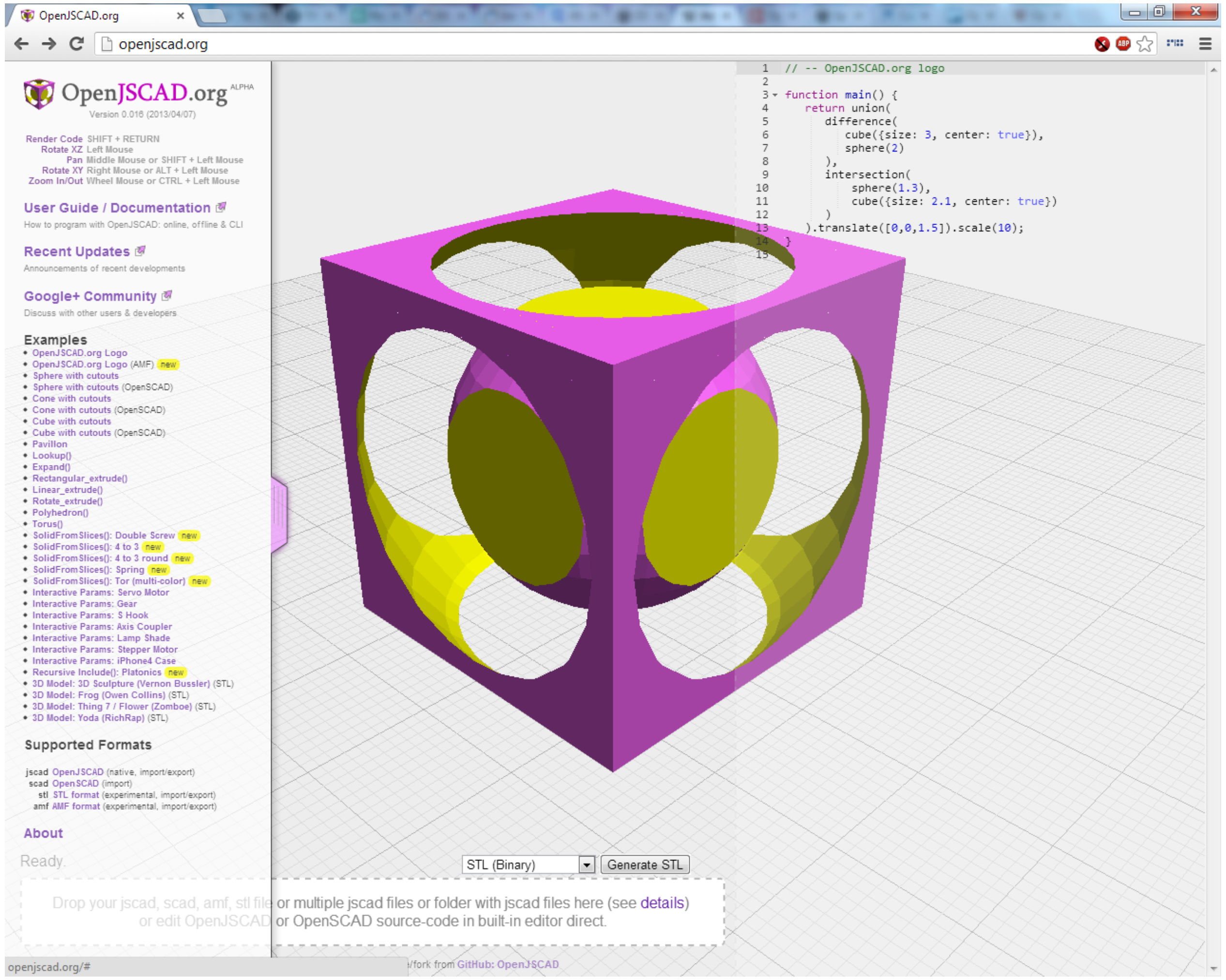
read on this documentation.

Web Browser & Command-Line Interface aka "Dual Use"

Web Browser (Online, Local & Offline)

OpenJSCAD.org contains the editor (Graphical User Interface) for you:


Web Browser
WebGL: Chrome & Firefox
http://openjscad.org
http://localhost/OpenJSCAD.org/
file:///OpenJSCAD.org/index.html
OpenJSCAD.org
OpenSCAD + JavaScript = OpenJSCAD
Supported File Formats:
JSCAD, SCAD*, STL, AMF*
SCAD: 95% functionality covered
AMF: uncompressed
Command Line Interface (CLI)
NodeJS: Linux, Windows, MacOS-X
% openjscad test.jscad



OpenJSCAD.org

openjscad.org

OpenJSCAD.org ALPHA
Version 0.016 (2013/04/07)

Render Code SHIFT + RETURN
Rotate XZ Left Mouse
Pan Middle Mouse or SHIFT + Left Mouse
Rotate XY Right Mouse or ALT + Left Mouse
Zoom In/Out Wheel Mouse or CTRL + Left Mouse

User Guide / Documentation
How to program with OpenJSCAD: online, offline & CLI

Recent Updates
Announcements of recent developments

Google+ Community
Discuss with other users & developers

Examples
- OpenJSCAD.org Logo
- OpenJSCAD.org Logo (AMF) new
- Sphere with cutouts
- Sphere with cutouts (OpenSCAD)
- Cone with cutouts
- Cone with cutouts (OpenSCAD)
- Cube with cutouts
- Cube with cutouts (OpenSCAD)
- Pavillon
- Lookup()
- Expand()
- Rectangular_extrude()
- Linear_extrude()
- Rotate_extrude()
- Polyhedron()
- Torus()
- SolidFromSlices(): Double Screw new
- SolidFromSlices(): 4 to 3 new
- SolidFromSlices(): 4 to 3 round new
- SolidFromSlices(): Spring new
- SolidFromSlices(): Tor (multi-color) new
- Interactive Params: Servo Motor
- Interactive Params: Gear
- Interactive Params: S Hook
- Interactive Params: Axis Coupler
- Interactive Params: Lamp Shade
- Interactive Params: Stepper Motor
- Interactive Params: iPhone4 Case
- Recursive Include(): Platonics new
- 3D Model: 3D Sculpture (Vernon Bussler) (STL)
- 3D Model: Frog (Owen Collins) (STL)
- 3D Model: Thing 7 / Flower (Zomboe) (STL)
- 3D Model: Yoda (RichRap) (STL)

Supported Formats

jscad	OpenJSCAD (native, import/export)
scad	OpenSCAD (import)
stl	STL format (experimental, import/export)
amf	AMF format (experimental, import/export)

About

Ready.

```
// -- OpenJSCAD.org logo

function main() {
   return union(
      difference(
         cube({size: 3, center: true}),
         sphere(2)
      ),
      intersection(
          sphere(1.3),
          cube({size: 2.1, center: true})
      )
   ).translate([0,0,1.5]).scale(10);
}
```

Drop your jscad, scad, amf, stl file or multiple jscad files or folder with jscad files here (see details)
or edit OpenJSCAD or OpenSCAD source-code in built-in editor direct.

openjscad.org/#

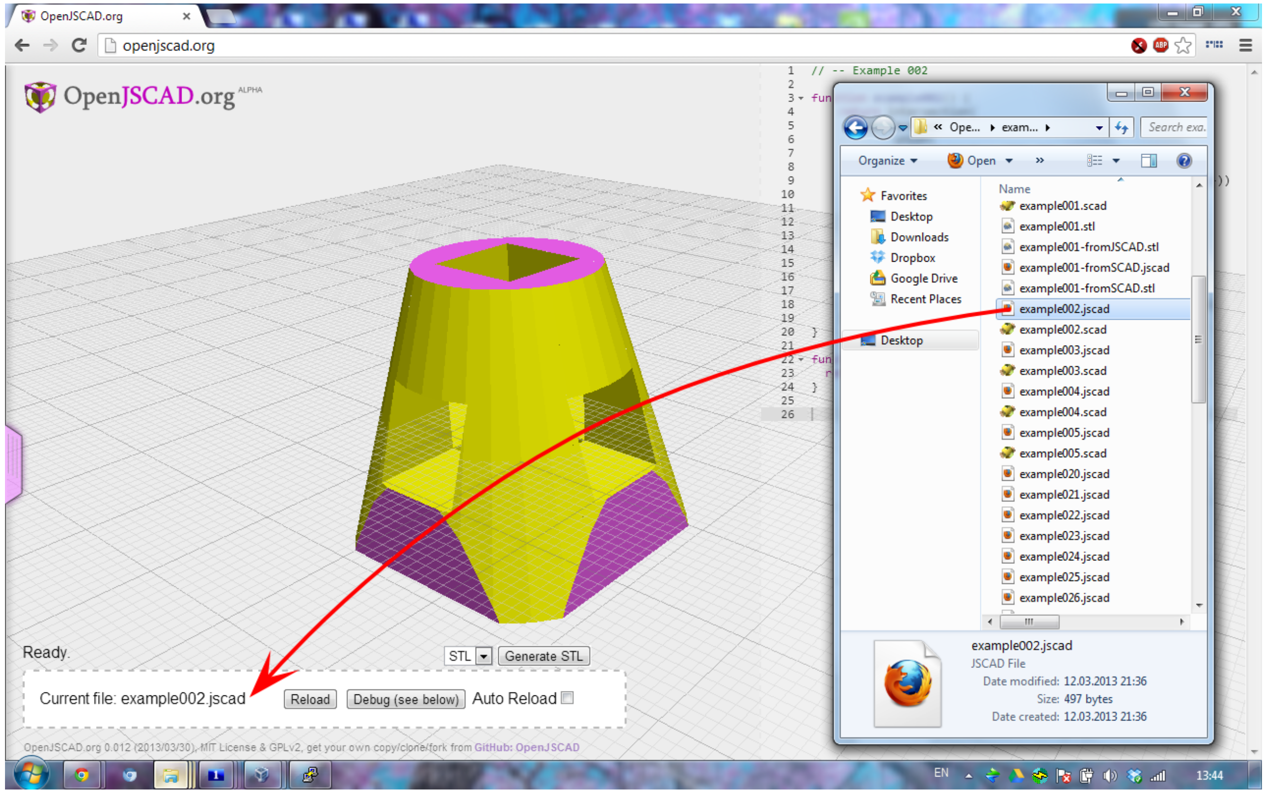
/fork from GitHub: OpenJSCAD

where you can

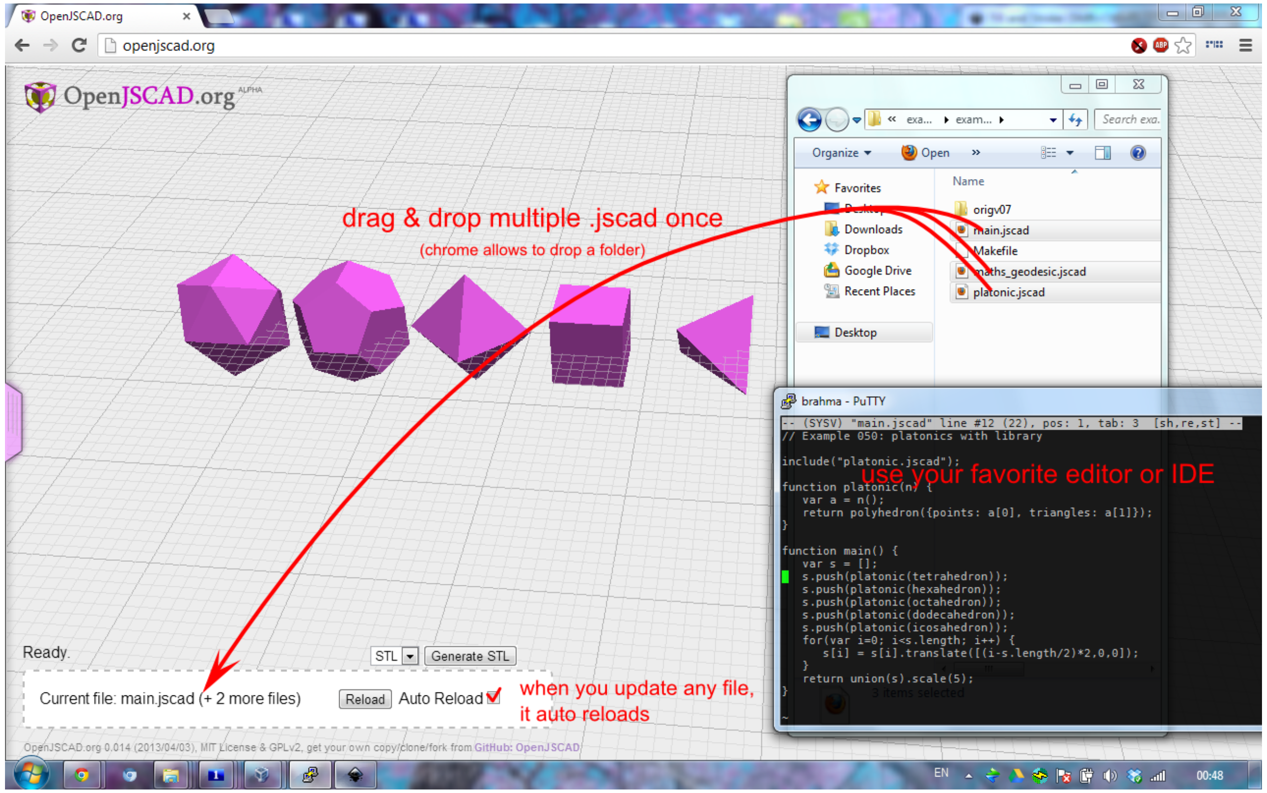
- edit online using the built-in editor, and also
- edit off-line via your favorite editor (just drag & drop the .jscad file(s) (Chrome & Firefox) or folder (Chrome only) into the area indicated in the browser, and make sure Auto Reload [x] is selected).

Clone this wiki locally

https://github.com/Spiritdu



Clone in Desktop


OpenJSCAD.org
openjscad.org
OpenJSCAD.org ALPHA
drag & drop multiple .jscad once
(chrome allows to drop a folder)
Organize
Open
Favorites
Desktop
Downloads
Dropbox
Google Drive
Recent Places
Desktop
Name
origv07
main.jscad
Makefile
maths_geodesic.jscad
platonic.jscad
brahma - PuTTY
use your favorite editor or IDE
Ready.
STL
Generate STL
Current file: main.jscad (+ 2 more files)
Reload
Auto Reload
when you update any file,
it auto reloads
OpenJSCAD.org 0.014 (2013/04/03), MIT License & GPLv2, get your own copy/clone/fork from GitHub: OpenJSCAD
EN
00:48

```
-- (SYSV) "main.jscad" line #12 (22), pos: 1, tab: 3  [sh,re,st] --
// Example 050: platonics with library

include("platonic.jscad");

function platonic(n) {
   var a = n();
   return polyhedron({points: a[0], triangles: a[1]});
}

function main() {
   var s = [];
   s.push(platonic(tetrahedron));
   s.push(platonic(hexahedron));
   s.push(platonic(octahedron));
   s.push(platonic(dodecahedron));
   s.push(platonic(icosahedron));
   for(var i=0; i<s.length; i++) {
      s[i] = s[i].translate([(i-s.length/2)*2,0,0]);
   }
   return union(s).scale(5);
}
```

Note: right now **Google Chrome** and **Firefox** are supported (requires WebGL), Opera and IE10 might follow.

Local Installation (Offline)

Prerequisites:

- install **NodeJS** (e.g. `apt-get install nodejs`), **be aware NodeJS >= 0.8.1** (Ubuntu 12.04 installs NodeJS 0.6.1).

Note: check how the actual NodeJS executable is named, either `/usr/local/bin/node` (default) or `/usr/bin/nodejs` and edit `openjscad` first line accordingly.

Assuming you cloned the depository:

```
% git clone git@github.com:Spiritdude/OpenJSCAD.org.git
% cd OpenJSCAD.org
% make install
```

Web Browser (Offline)

Access the local copy of `index.html` with your browser (**Google Chrome** or **Firefox** for now, Opera and IE10 will follow).

Note: You can only drag & drop multiple .jscad files, but not a folder as such (regardless of Chrome or Firefox)

Command-Line Interface (Offline)

```
% cd examples
% openjscad logo.jscad
% openjscad logo.jscad -o test.stl
% openjscad logo.jscad -of stl
```

which creates `logo.stl` or `test.stl` .

Additional you can import OpenSCAD (.scad), STL ASCII or Binary (.stl) and AMF (.amf) files, and create .jscad, .stl (ASCII or Binary), dxf or .amf:

```
% openjscad example001.scad -o example001.jscad
% openjscad example001.scad -o example001.stl
% openjscad frog.stl -o frog.jscad
% openjscad frog.stl -o frog2.stl             # does actually stl -> jscad -> stl (ascii)
% openjscad example001.jscad -o example001.amf
% openjscad example001.jscad -of amf          # creates example001.amf
% openjscad example001.jscad -of stlb         # creates example001.stl (binary)
% openjscad example001.jscad -of stlb -o test.stl
```

- `-o` stands for output
- `-of` stands for output format (jscad, stl (default), stla, stlb, amf, dxf)

See also how to pass variables from CLI the to main(): Command Line Interface Parameters.

Language / File Format Support

Currently following languages and file-formats are supported:

- **JSCAD** / .jscad (OpenJSCAD.org): this document describes the details of the .jscad, which is essentially JavaScript + CSG and 3D functions and methods to construct solids
- **SCAD** / .scad (OpenSCAD.org): OpenSCAD User Manual describes the .scad syntax, apprx. 95% of the language is supported (see OpenSCAD Import Section)
- **STL** / .stl: Wikipedia: STL (file format)
- **AMF** / .amf: Additive Manufacturing File Format (very experimental)

When you drag & drop files, the language or file-format is set according the file-extension (.jscad, .scad, .stl, .amf). When you start to edit direct in the Web GUI (browser), the default language is **JSCAD .jscad**.

Direct Link for Local or Remote JSCAD, SCAD, STL and AMF

You can give other people a link to a specific example:

Local

- **http://openjscad.org/#examples/slices/tor.jscad**
- **http://localhost/OpenJSCAD.org/#examples/slices/tor.jscad**

In case you reference a JSCAD, the **include()** is supported.

Remote

- **http://openjscad.org/#http://openjscad.org/examples/slices/tor.jscad**
- **http://openjscad.org/#http://www.thingiverse.com/download:164128** (STL)
- **http://openjscad.org/#http://pastebin.com/raw.php?i=5RbzVguT** (SCAD)
- **http://openjscad.org/#http://amf.wikispaces.com/file/view/Rook.amf/268219202/Rook.amf** (AMF)

In case you reference a .JSCAD, the **include()** is not supported (perhaps later).

Anatomy of a JSCAD File

An OpenJSCAD .jscad file has to have at least one function defined, the `main()` , unless it's a library or helper (multiple files), which has to return a CSG object or an array of non-intersecting CSG objects:

```
function main() {
   return union(sphere(), ...);    // an union of objects or
   return [sphere(), ...];        // an array of non-intersecting objects
}
```

or like this:

```
var w = new Array();
function a() {
   w.push( sphere() );
   w.push( cube().translate([2,0,0]) );
}
function main() {
   a();
   return w;
}
```

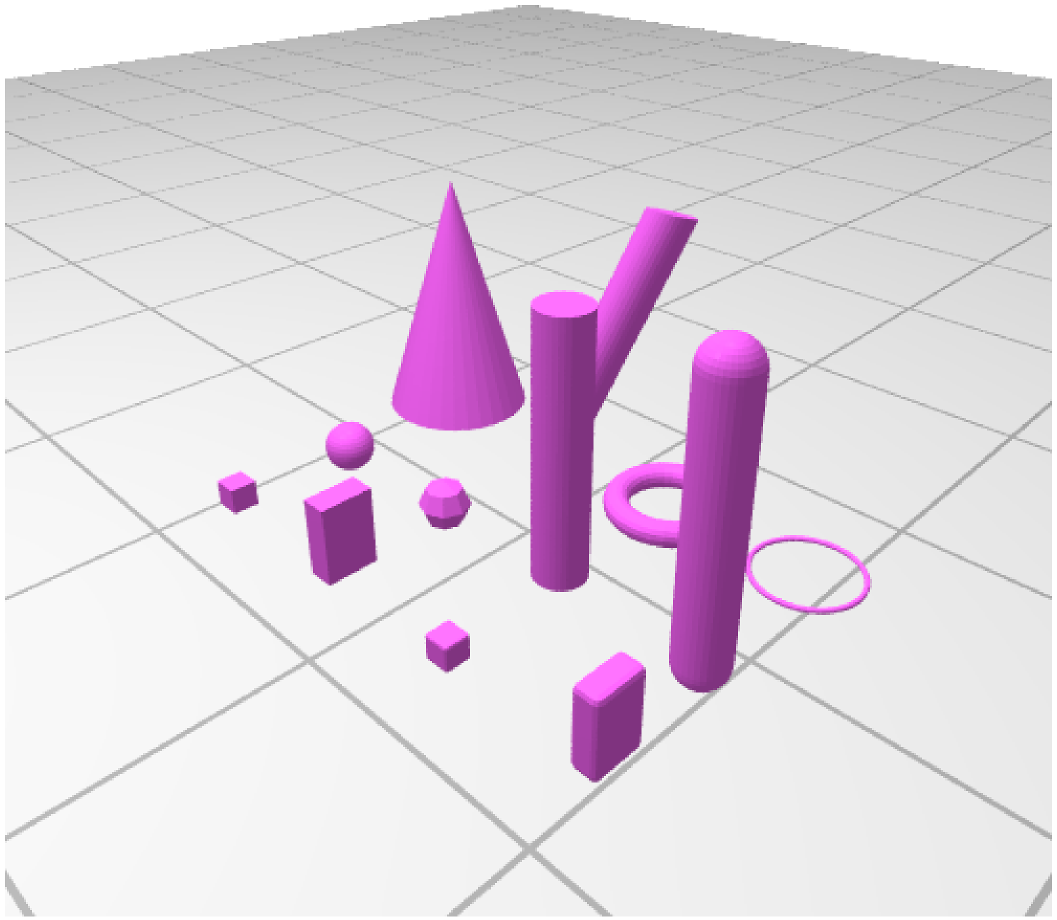
but this does **not work**:

```
var w = new Array();
w.push( sphere() );                    // Note: it's not within a function (!!)
w.push( cube().translate([2,0,0]) );

function main() {
    return w;
}
```

as all CSG creations, like 3D primitives, have to occur within functions which are called eventually through `main()` .

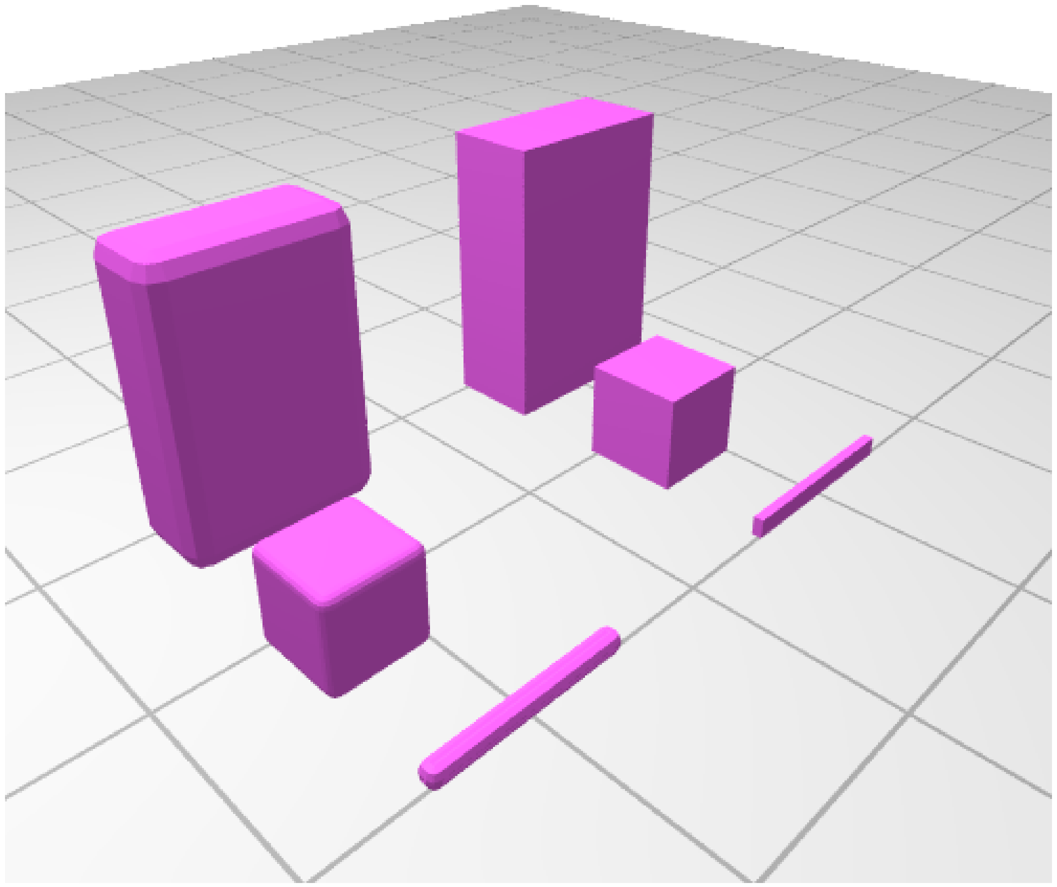
3D Primitives

The parameters are passed in an object; most parameters are optional. 3D vectors can be passed in an array. If a scalar is passed for a parameter which expects a 3D vector, it is used for the x, y and z value. In other words: radius: 1 will give radius: [1,1,1].



All rounded solids have a 'fn' or 'resolution' parameter which controls tesselation. If resolution is set to 8, then 8 polygons per 360 degree of revolution are used. Beware that rendering time will increase dramatically when increasing the resolution. For a sphere the number of polygons increases quadratically with the resolution used. If the resolution parameter is omitted, the following two global defaults are used: CSG.defaultResolution2D and CSG.defaultResolution3D. The former is used for 2D curves (circle, cylinder), the latter for 3D curves (sphere, 3D expand).

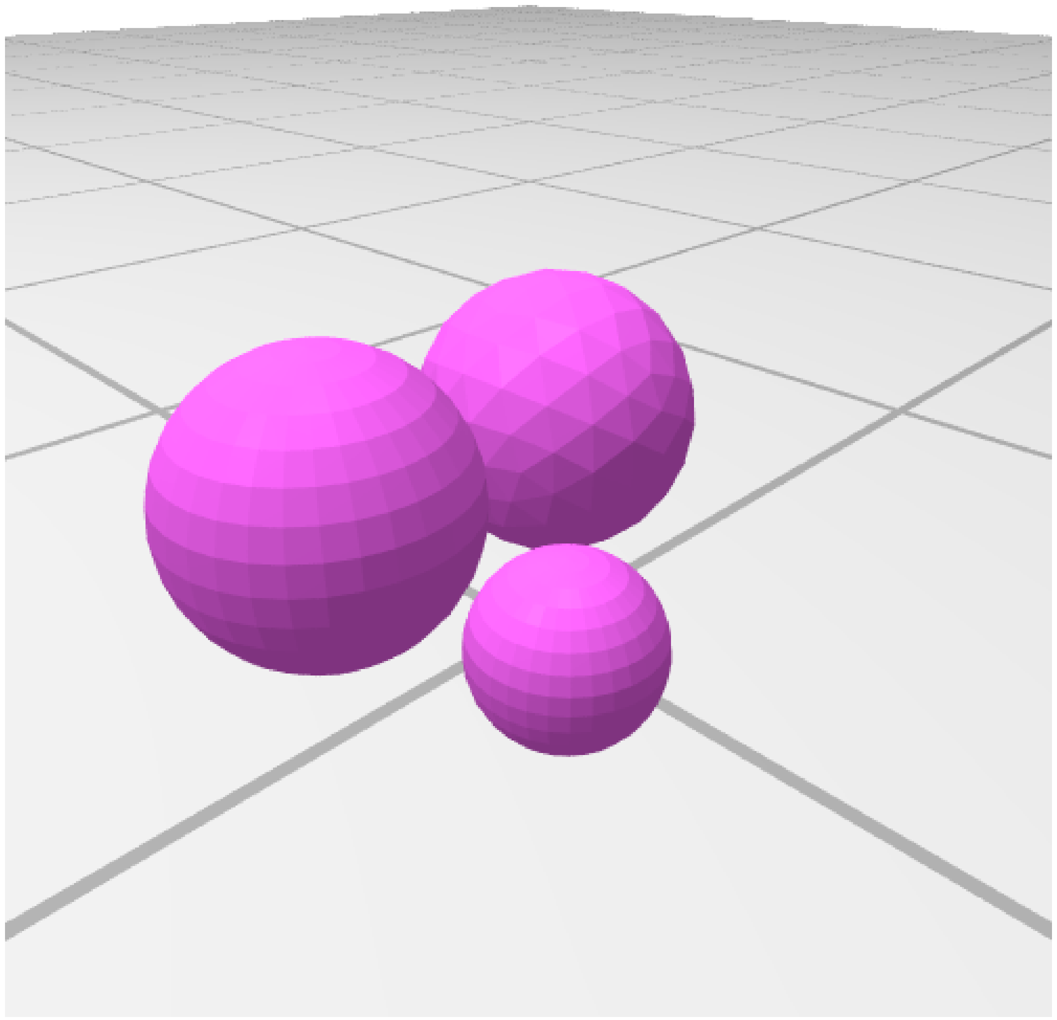
Cube



Cube or rather boxes can be created like this:

```
cube();                             // openscad like
cube(1);
cube({size: 1});
cube({size: [1,2,3]});
cube({size: 1, center: true});     // default center:false
cube({size: 1, center: [true,true,false]}); // individual axis center true or false
cube({size: [1,2,3], round: true});

CSG.cube();                         // object-oriented
CSG.cube({
  center: [0, 0, 0],
  radius: [1, 1, 1]
});
CSG.roundedCube({                   // rounded cube
  center: [0, 0, 0],
  radius: 1,
  roundradius: 0.2,
  resolution: 8,
});
```

Sphere



Spheres can be created like this:

```
sphere();                          // openscad like
sphere(1);
sphere({r: 2});                    // Note: center:true is default (unlike other primitiv
es, as OpenSCAD)
sphere({r: 2, center: false});     // Note: OpenSCAD doesn't support center for sphere bu
```

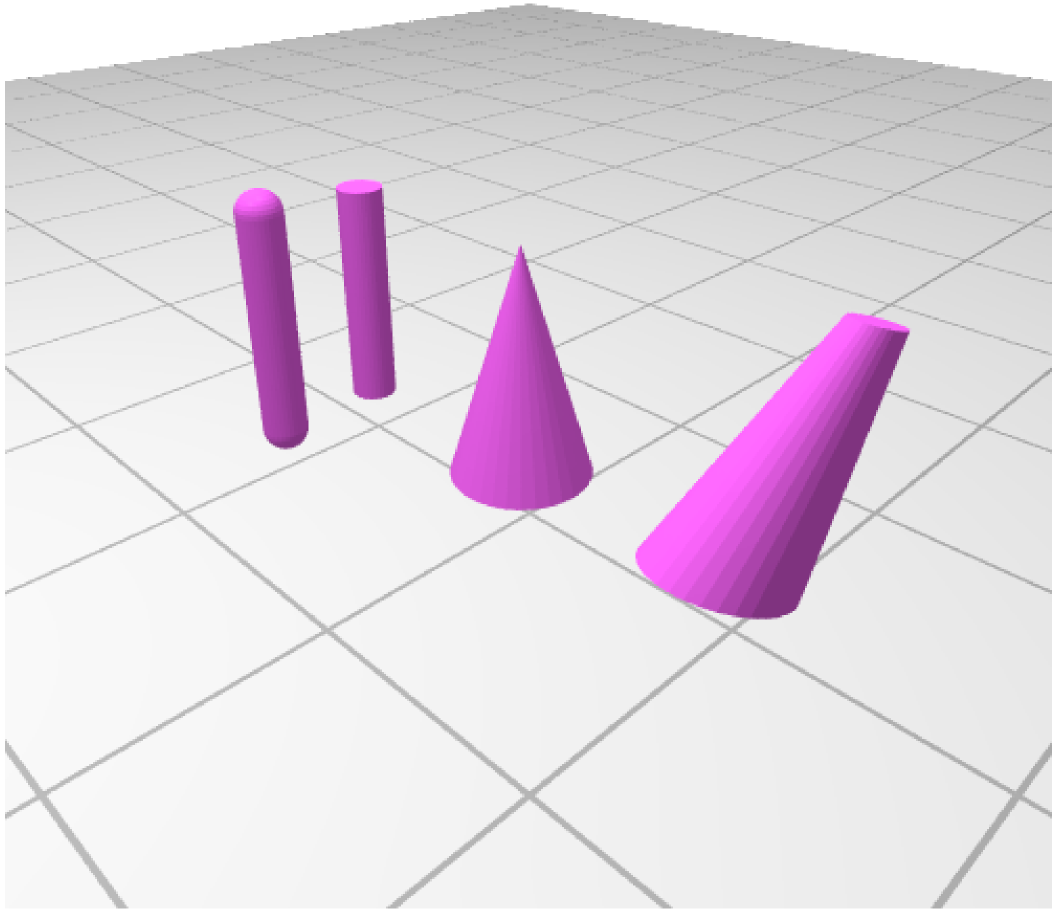
```
t we do
sphere({r: 2, center: [true, true, false]}); // individual axis center
sphere({r: 10, fn: 100 });
sphere({r: 10, fn: 100, type: 'geodesic'});  // geodesic approach (icosahedron further tr
iangulated)

CSG.sphere();                      // object-oriented
CSG.sphere({
  center: [0, 0, 0],
  radius: 2,                      // must be scalar
  resolution: 32
});
```

whereas fn is the amount of segments to approximate a sphere (default 32, total polygons per sphere fn*fn).

In case of `type: 'geodesic'` the fn tries to match the non-geodesic fn, yet, actually changes in steps of 6 (e.g. fn=6..11 is the same), fn = 1 reveals the base form: the icosahedron.

Note: creating sphere(s) and then operate with them (e.g. union(), intersection() etc) slows down rendering / construction procedure due the large amount of polygons in use.

Cylinder

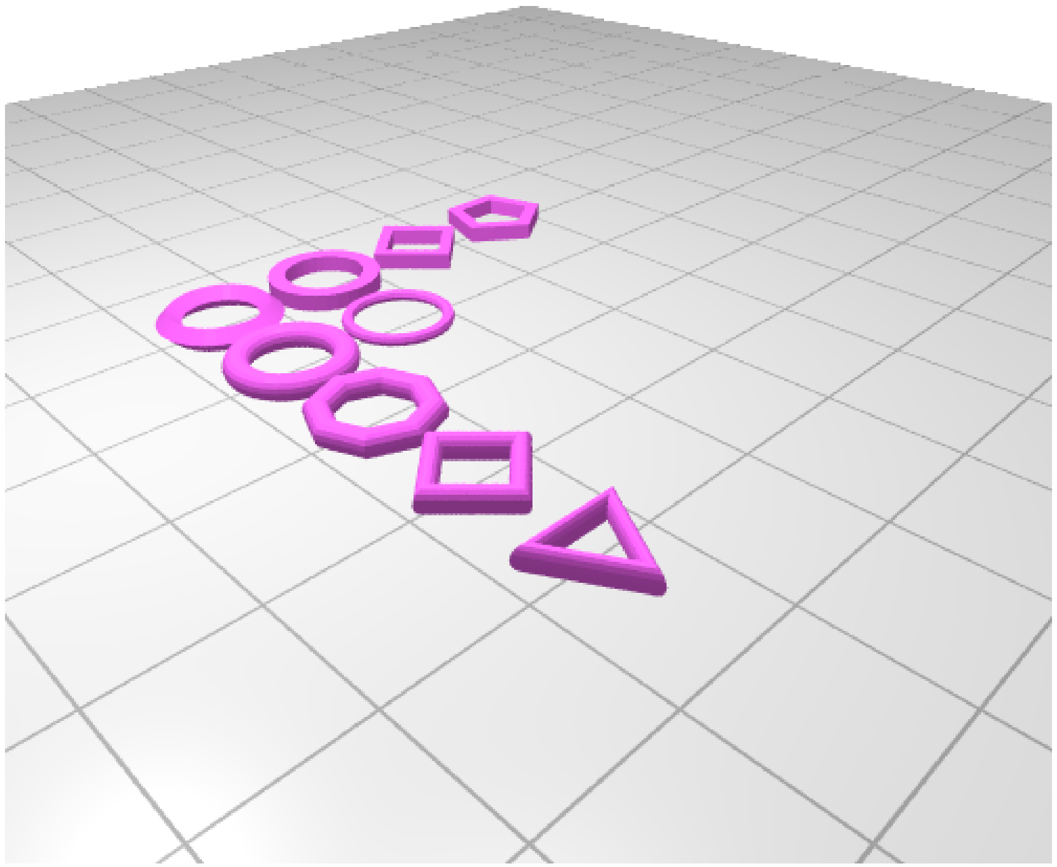


Cylinders and cones can be created like this:

```
cylinder({r: 1, h: 10});                 // openscad like
cylinder({d: 1, h: 10});
cylinder({r: 1, h: 10, center: true});   // default: center:false
cylinder({r: 1, h: 10, center: [true, true, false]});  // individual x,y,z center flags
cylinder({r: 1, h: 10, round: true});
cylinder({r1: 3, r2: 0, h: 10});
cylinder({d1: 1, d2: 0.5, h: 10});
cylinder({start: [0,0,0], end: [0,0,10], r1: 1, r2: 2, fn: 50});

CSG.cylinder({                      // object-oriented
  start: [0, -1, 0],
  end: [0, 1, 0],
  radius: 1,                        // true cylinder
  resolution: 16
});
CSG.cylinder({
  start: [0, -1, 0],
  end: [0, 1, 0],
  radiusStart: 1,                   // start- and end radius defined, partial cones
  radiusEnd: 2,
  resolution: 16
});
CSG.roundedCylinder({               // and its rounded version
  start: [0, -1, 0],
  end: [0, 1, 0],
  radius: 1,
  resolution: 16
});
```

whereas fn is the amount of segments to approximate the circular profile of the cylinder (default 32).

Torus



A torus is defined as such:

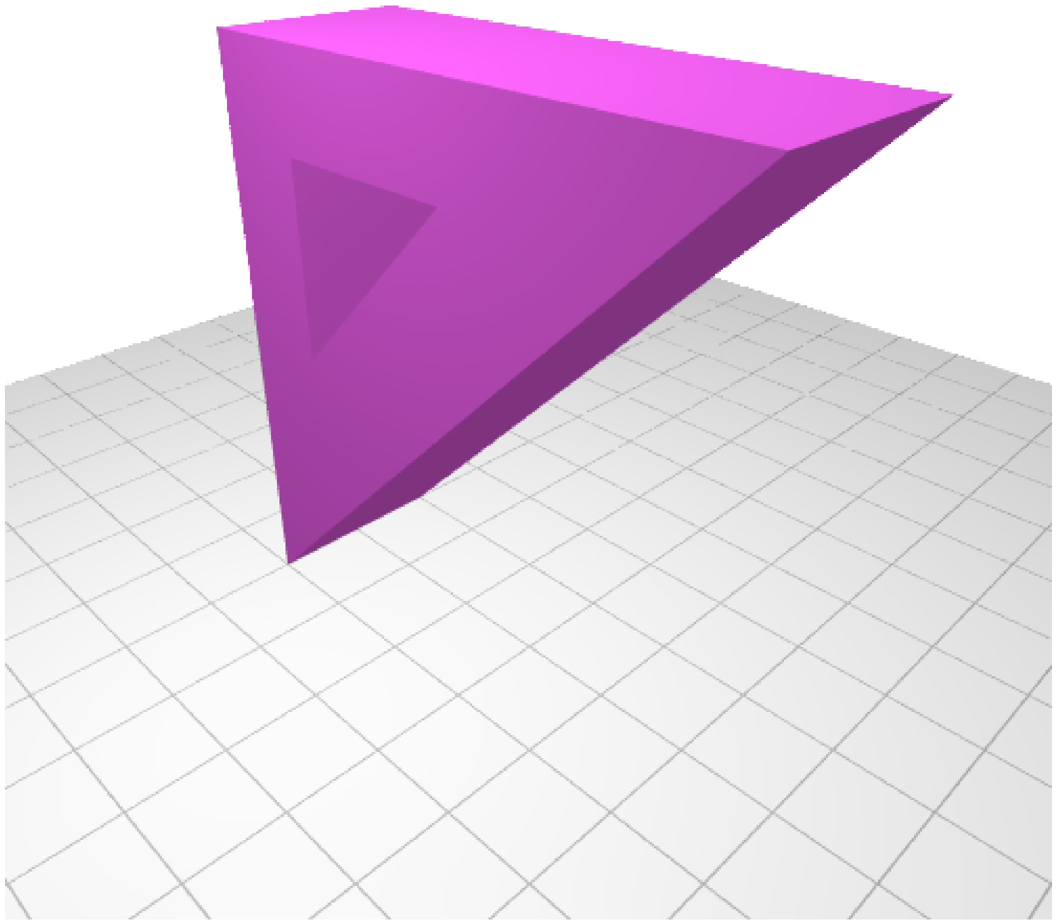
- ri = inner radius (default: 1),
- ro = outer radius (default: 4),
- fni = inner resolution (default: 16),
- fno = outer resolution (default: 32),

- roti = inner rotation (default: 0)

```
torus();                    // ri = 1, ro = 4;
torus({ ri: 1.5, ro: 3 });
torus{{ ri: 0.2 });

torus({ fni:4 });           // make inner circle fn = 4 => square
torus({ fni:4,roti:45 });   // rotate inner circle, so flat is top/bottom
torus({ fni:4,fno:4,roti:45 });
torus({ fni:4,fno:5,roti:45 });
```

Polyhedron

Create a polyhedron with a list of points and a list of triangles or polygons. The point list is all the vertexes of the shape, the triangle list is how the points relates to the surfaces of the polyhedron:



```
polyhedron({      // openscad-like (e.g. pyramid)
  points: [ [10,10,0],[10,-10,0],[-10,-10,0],[-10,10,0], // the four points at base
            [0,0,10] ],                                  // the apex point
  triangles: [ [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
});
```

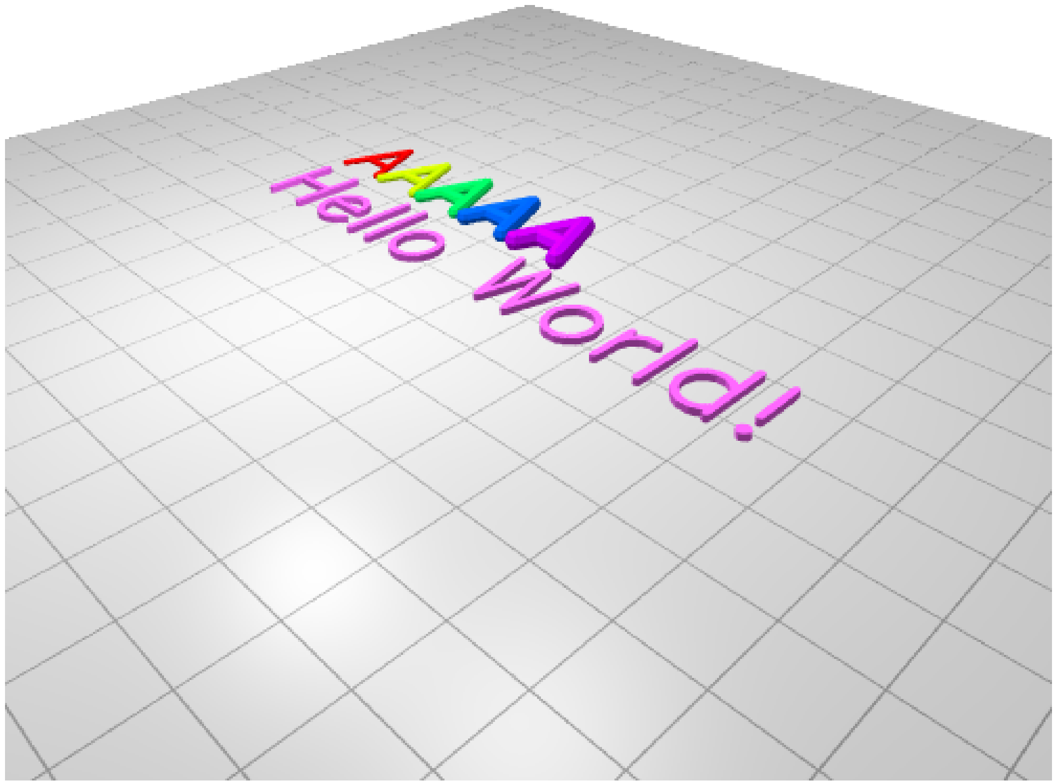
Additionally you can also define `polygons: [ [0,1,4,5], [..] ]` too, not just `triangles:`.

You can also create a polyhedron at a more low-level and object-oriented:

```
var polygons = [];
polygons.push(new CSG.Polygon([
      new CSG.Vertex(new CSG.Vector3D(x1,y1,z1)),
      new CSG.Vertex(new CSG.Vector3D(x2,y2,z2)),
      new CSG.Vertex(new CSG.Vector3D(x3,y3,z3))
   ])
);
// add more polygons and finally:
solid = CSG.fromPolygons(polygons);
```

Text

`vector_text(x,y,string)` and `vector_char(x,y,char)` give you line segments of a text or character rendered in vector:



```
var l = vector_text(0,0,"Hello World!");   // l contains a list of polylines to be drawn
var o = [];
l.forEach(function(pl) {                   // pl = polyline (not closed)
   o.push(rectangular_extrude(pl, {w: 2, h: 2}));   // extrude it to 3D
});
return union(o);
```

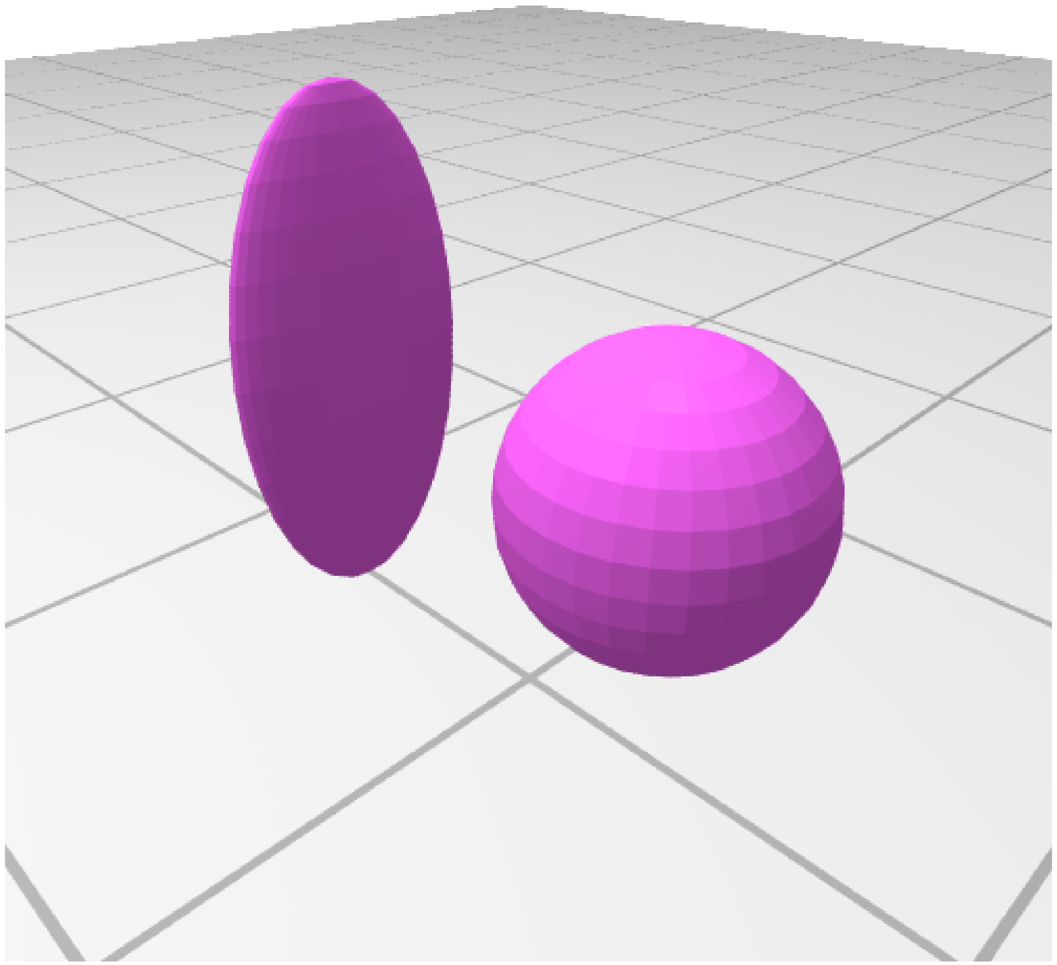
Also multiple-line with "\n" is supported, for now just left-align is supported. If you want to dive more into the details, you can request a single character:

```
var c = vector_char(x,y,"A");
c.width;    // width of the vector font rendered character
c.segments; // array of segments / polylines
```

3D Transformations

Scale

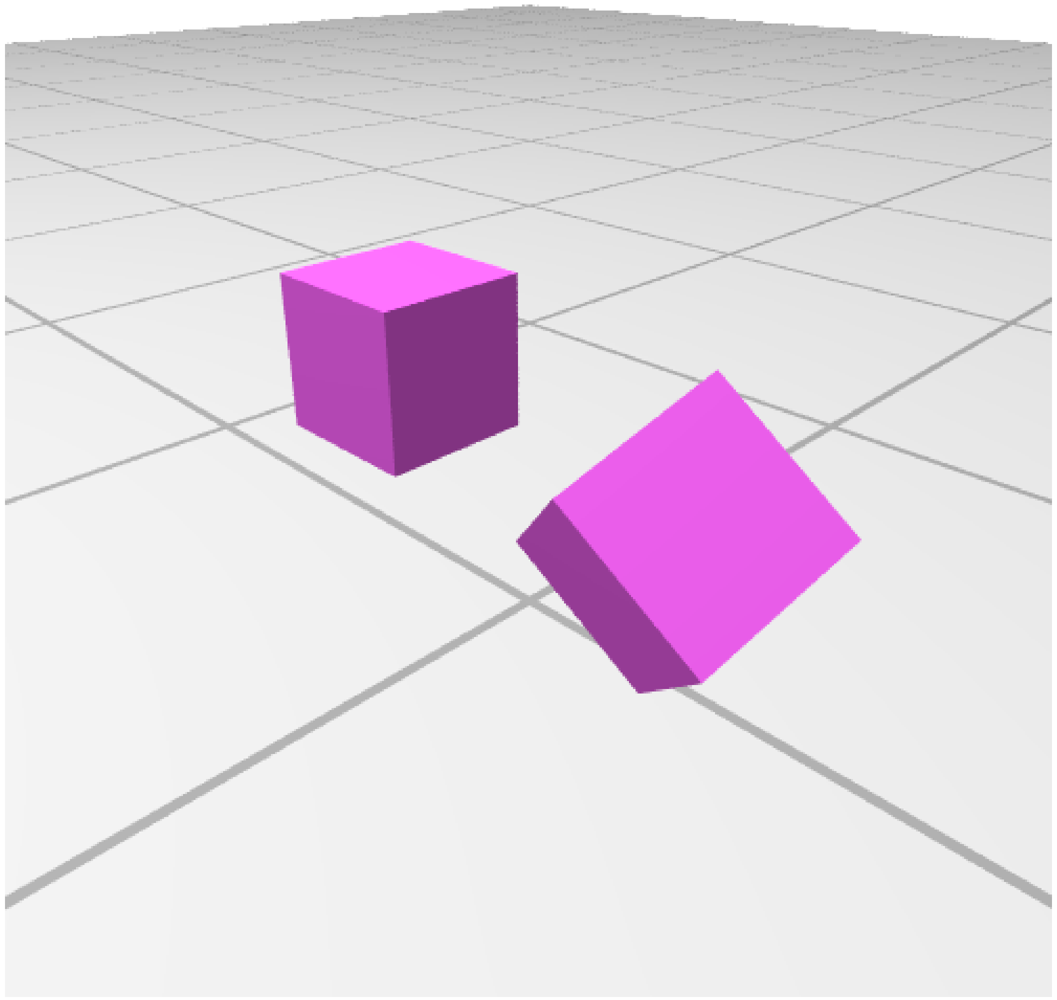
```
scale(2,obj);          // openscad like
scale([1,2,3],obj);    //      ''

obj.scale([1,2,3]);    // object-oriented
```



Rotate

```
rotate([90,15,30],obj);       // openscad like
rotate(90,[1,0.25,0.5],obj);  //    ''

obj.rotateX(90);              // object-oriented
```

```
    obj.rotateY(45);
    obj.rotateZ(30);
```



Translate

```
translate([0,0,10],obj);  // openscad like

obj.translate([0,0,10]);  // object-oriented
```

Center

Centering an object altogether or axis-wise:

```
center(true,cube());                // openscad-like all axis
center([true,true,false],cube());   // openscad-like axis-wise [x,y,z]

cube().center(true);                // object oriented center all axis
cube().center([true,false,true]);   // object oriented center axis-wise [x,y,z]
```

false = do nothing, true = center axis

It center() and .center() helps you to compose a symmetric whose complete size you don't know when composing, e.g. from parametric design.

Matrix Operations

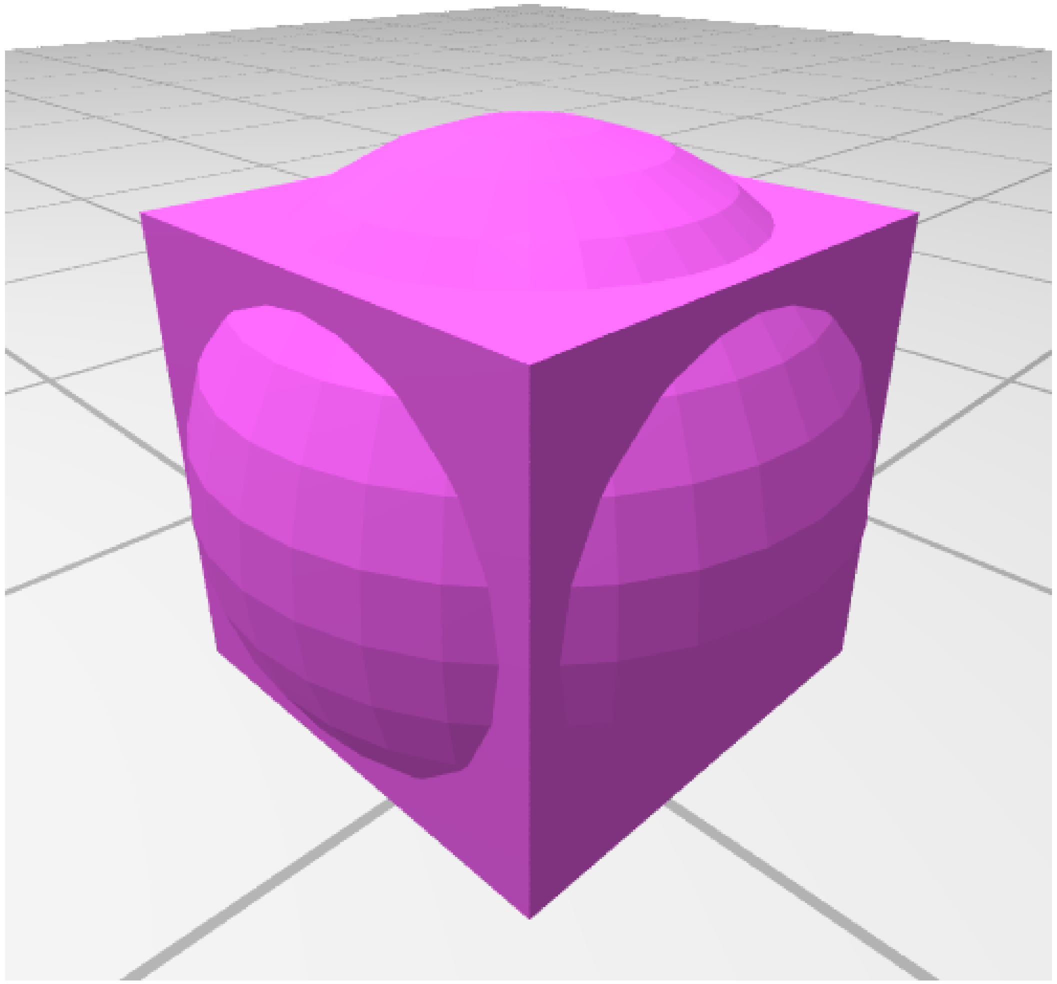
```
var m = new CSG.Matrix4x4();
m = m.multiply(CSG.Matrix4x4.rotationX(40));
m = m.multiply(CSG.Matrix4x4.rotationZ(40));
m = m.multiply(CSG.Matrix4x4.translation([-.5, 0, 0]));
m = m.multiply(CSG.Matrix4x4.scaling([1.1, 1.2, 1.3]));

// and apply the transform:
```

```
var cube3 = cube.transform(m);
```

Mirror

```
mirror([10,20,90], cube(1)); // openscad like

var cube = CSG.cube().translate([1,0,0]);   // object-oriented

var cube2 = cube.mirroredX(); // mirrored in the x=0 plane
var cube3 = cube.mirroredY(); // mirrored in the y=0 plane
var cube4 = cube.mirroredZ(); // mirrored in the z=0 plane

// create a plane by specifying 3 points:
var plane = CSG.Plane.fromPoints([5,0,0], [5, 1, 0], [3, 1, 7]);

// and mirror in that plane:
var cube5 = cube.mirrored(plane);
```

Union

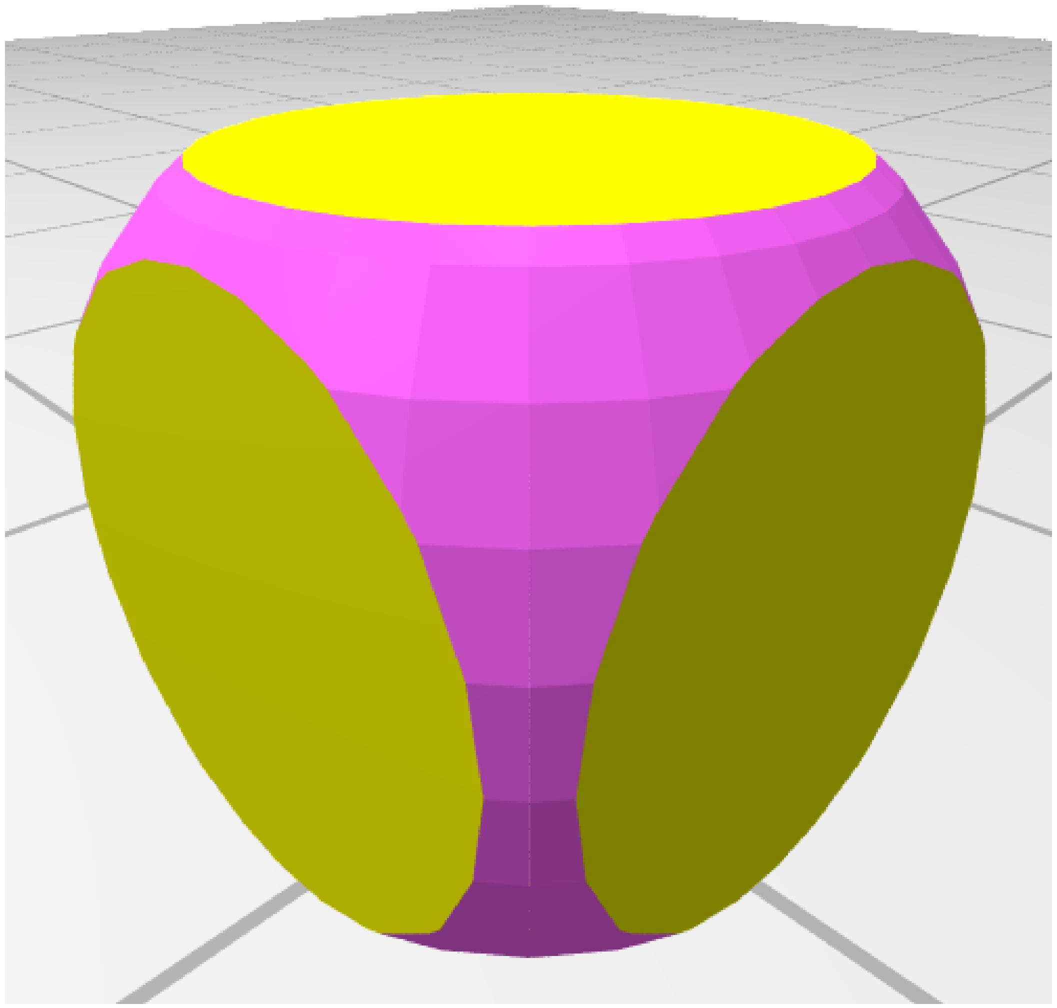


```
union(sphere({r: 1, center:true}),cube({size: 1.5, center:true}));  // openscad like
```

multiple objects can be added, also arrays.

```
sphere({r: 1, center:true}).union(cube({size: 1.5, center:true}));  // object-oriented
```

Intersection



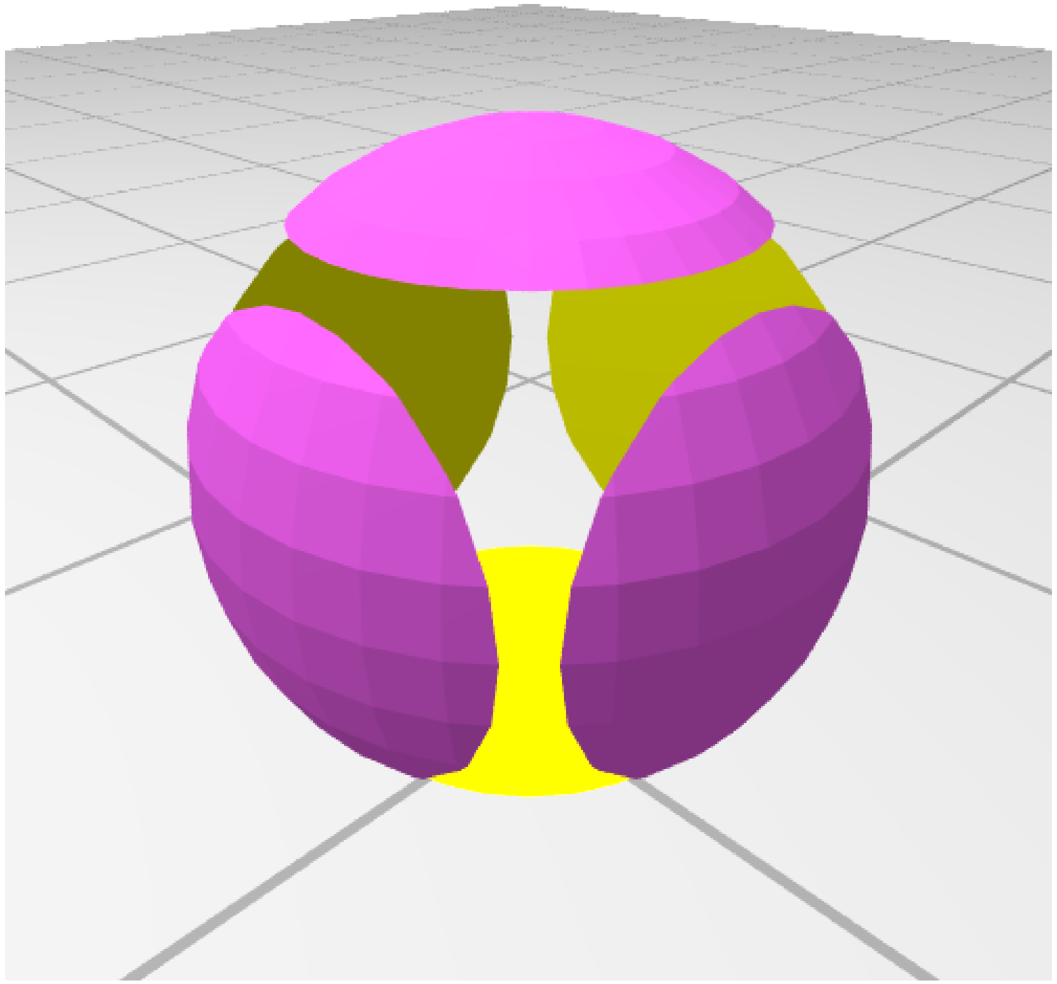
```
intersection(sphere({r: 1, center:true}),cube({size: 1.5, center:true})); // openscad lik
e
```

multiple objects can be intersected, also arrays.

```
sphere({r: 1, center:true}).intersect(cube({size: 1.5, center:true}));   // object-orient
ed
```

Note: intersection() (openscad like) vs intersect() (method, object-oriented)

Difference (Substraction)



```
difference(sphere({r: 1, center:true}),cube({size: 1.5, center:true}));    // openscad li
ke
```

multiple objects can be differentiated (subtracted) from the first element, also arrays.

```
sphere({r: 1, center:true}).subtract(cube({size: 1.5, center:true}));      // object-orie
nted
```

Note: difference() (openscad like) vs substract() (method, object-oriented)

2D Primitives

Circle

```
circle();                        // openscad like
circle(1);
circle({r: 2, fn:5});
circle({r: 3, center: true});    // center: false (default)

CAG.circle({center: [0,0], radius: 3, resolution: 32});   // object-oriented
```

Square / Rectangle

```
square();                                   // openscad like
square(1);                                  // 1x1
square([2,3]);                              // 2x3
square({size: [2,4], center: true});        // 2x4, center: false (default)

CAG.rectangle({center: [0,0], radius: [w/2, h/2]});   // object-oriented, whereas w or h
= side-length of square
CAG.roundedRectangle({center: [0,0], radius: [w/2, h/2], roundradius: 1, resolution: 4});
```

Polygon

```
polygon([ [0,0],[3,0],[3,3] ]);                // openscad like
polygon({ points: [ [0,0],[3,0],[3,3] ] });
polygon({ points: [ [0,0],[3,0],[3,3],[0,6] ], paths: [ [0,1,2],[1,2,3] ] }); // multiple
 paths not yet implemented

var shape1 = CAG.fromPoints([ [0,0],[5,0],[3,5],[0,5] ]);    // object-oriented
```

2D Transformations

```
translate([2,2], circle(1));      // openscad like
rotate([0,0,90], square());       //     ''
shape = center(true, shape());    // center both axis
shape = center([true,false], shape()); // center axis-wise [x,y]

shape = shape.translate([-2, -2]);   // object-oriented
shape = shape.rotateZ(20);
shape = shape.scale([0.7, 0.9]);
shape = shape.center(true);          // center both axis
scape = shape.center([true,false]);  // center axis-wise [x,y]
```

2D Paths

A path is simply a series of points, connected by lines. A path can be open or closed (an additional line is drawn between the first and last point). 2D paths are supported through the CSG.Path2D class. The difference between a 2D Path and a 2D CAG is that a path is a 'thin' line, whereas a

CAG is an enclosed area.

Paths can be contructed either by giving a series of 2D coordinates, or through the CSG.Path2D.arc() function, which creates a circular curved path. Paths can be concatenated, the result is a new path.

A path can be converted to a CAG in two ways:

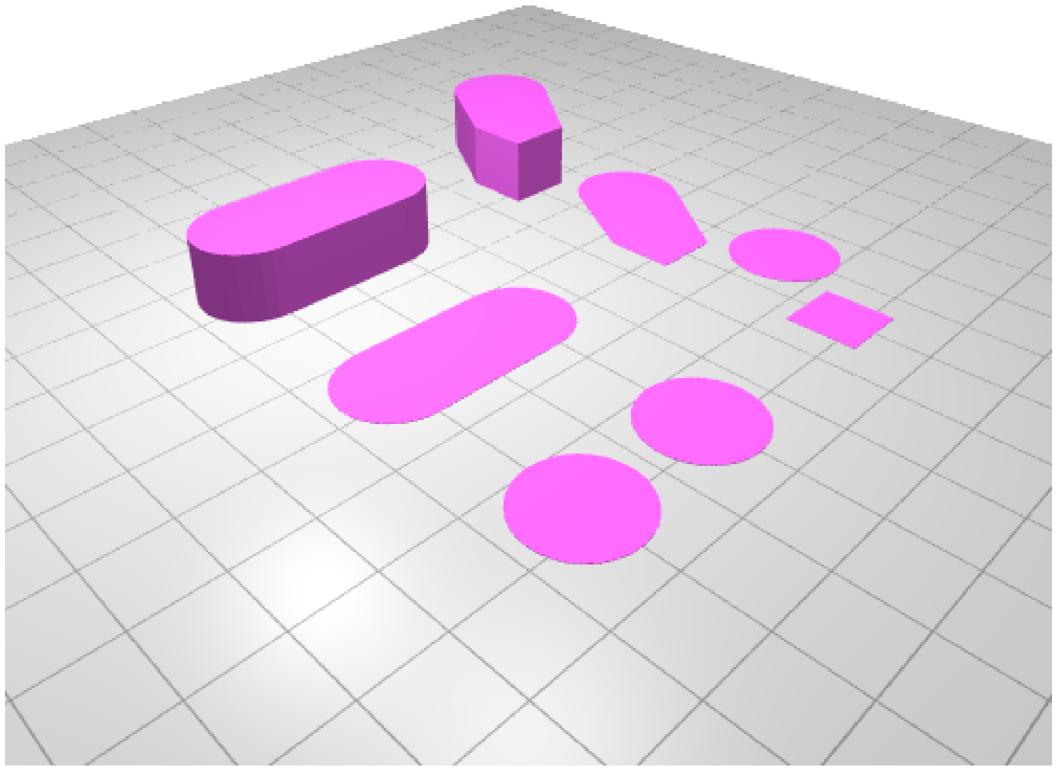
- expandToCAG(pathradius, resolution) traces the path with a circle, in effect making the path's line segments thick.
- innerToCAG() creates a CAG bounded by the path. The path should be a closed path.

Creating a 3D solid is currently supported by the rectangularExtrude() function. This creates a 3D shape by following the path with a 2D rectangle (upright, perpendicular to the path direction):

```
var path = new CSG.Path2D([ [10,10], [-10,10] ], /* closed = */ false);
var anotherpath = new CSG.Path2D([ [-10,-10] ]);
path = path.concat(anotherpath);
path = path.appendPoint([10,-10]);
path = path.close(); // close the path

// of course we could simply have done:
// var path = new CSG.Path2D([ [10,10], [-10,10], [-10,-10], [10,-10] ], /* closed = */ t
rue);

// We can make arcs and circles:
var curvedpath = CSG.Path2D.arc({
  center: [0,0,0],
  radius: 10,
  startangle: 0,
  endangle: 180,
  resolution: 16,
  });
```

Hull

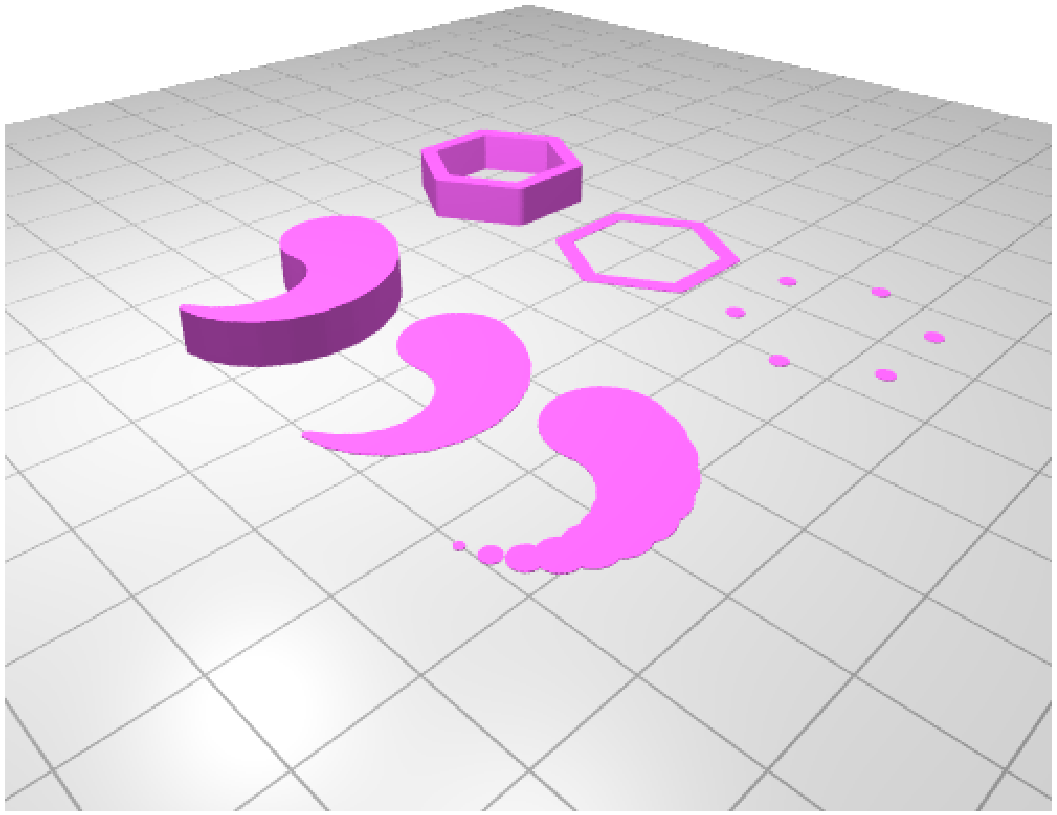
You can convex hull multiple 2D polygons (e.g. circle(), square(), polygon()) together:



```
var h = hull( square(10),circle(10).translate([10,10,0]) );

linear_extrude({ height: 10 }, h);
```

Chain Hull

Chained hulling is a variant of hull on multiple 2D forms, essentially sequential hulling and then union those, based on an idea by Whosa whatsis:

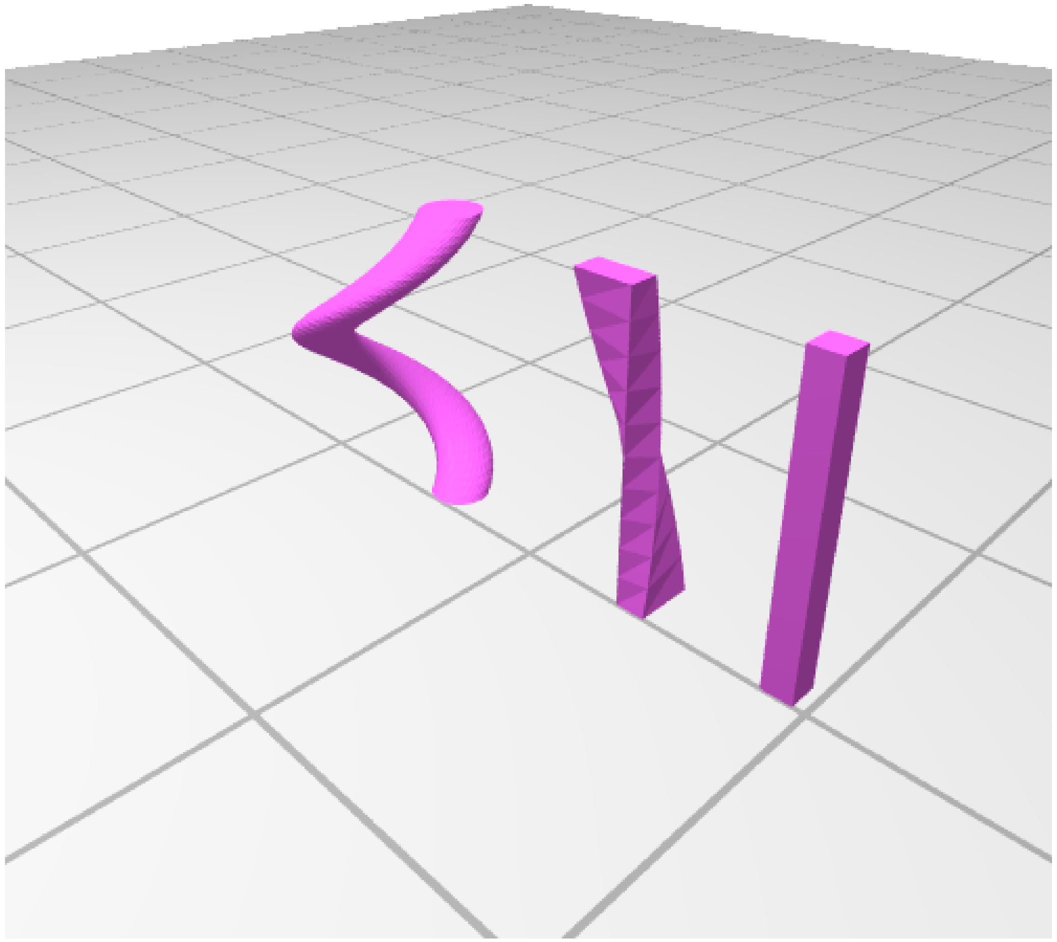


```
chain_hull(
    circle(), circle().translate([2,0,0]), ... );   // list of CAG/2D forms

var a = [];
a.push(circle());
chain_hull( a );                           // array of CAG/2D forms
chain_hull({closed: true},             // default is false
   circle(),circle().translate([2,0,0]),circle().translate([2,2,0]));
```

Extruding / Extrusion

Linear Extrude

Extruding 2D shapes into 3D, given height, twist (degrees), and slices (if twist is made):



```
// openscad like
linear_extrude({ height: 10 }, square());
linear_extrude({ height: 10, twist: 90 }, square([1,2]));
linear_extrude({ height: 10, twist: 360, slices: 50}, circle().translate([1,0,0]) );

linear_extrude({ height: 10, center: true, twist: 360, slices: 50}, translate([2,0,0], sq
uare([1,2])) );
```

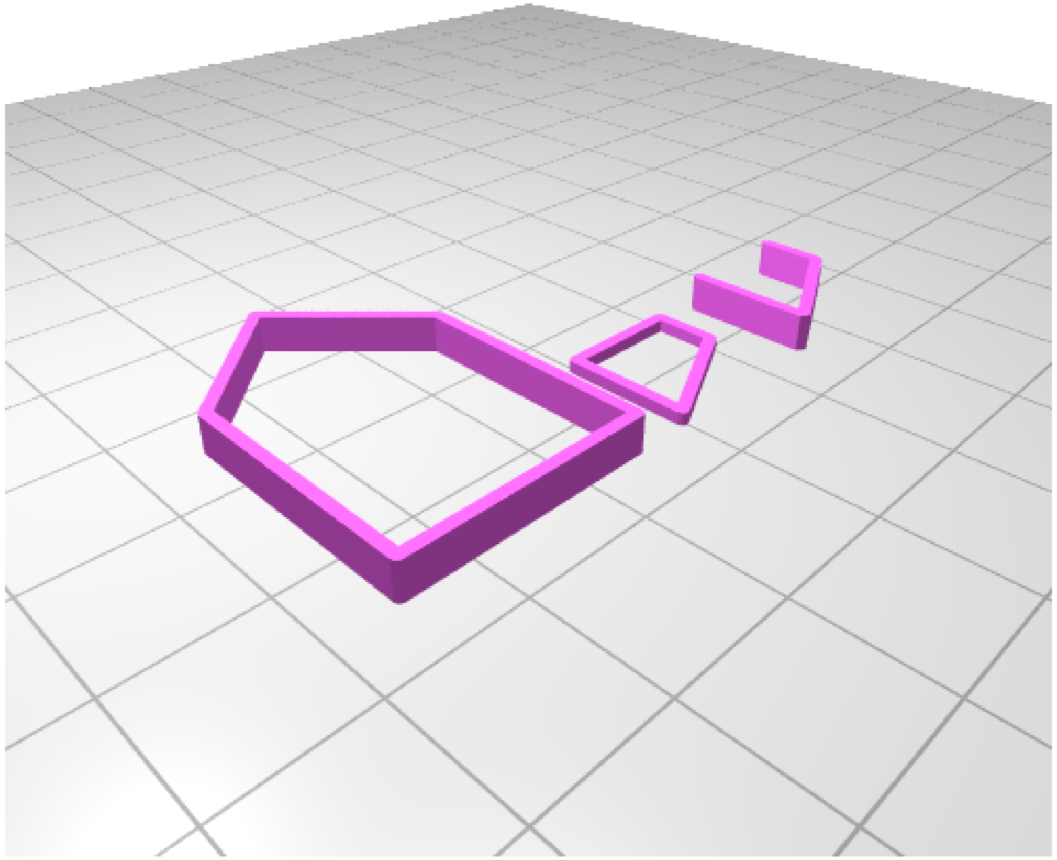
```
linear_extrude({ height: 10, center: true, twist: 360, slices: 50}, square([1,2]).transla
te([2,0,0]) );
```

Linear extrusion of 2D shape, with optional twist. The 2d shape is placed in in z=0 plane and extruded into direction <offset> (a CSG.Vector3D). The final face is rotated degrees. Rotation is done around the origin of the 2d shape (i.e. x=0, y=0) twiststeps determines the resolution of the twist (should be >= 1), returns a CSG object:

```
// object-oriented
var c = CAG.circle({radius: 3});
extruded = c.extrude({offset: [0,0,10], twistangle: 360, twiststeps: 100});
```

Rectangular Extrude

Extrude the path by following it with a rectangle (upright, perpendicular to the path direction), returns a CSG solid.



Simplified (openscad like, even though OpenSCAD doesn't provide this) via rectangular_extrude(), where as

- w: width (default: 1),
- h: height (default: 1),
- fn: resolution (default: 8), and

- closed: whether path is closed or not (default: false)

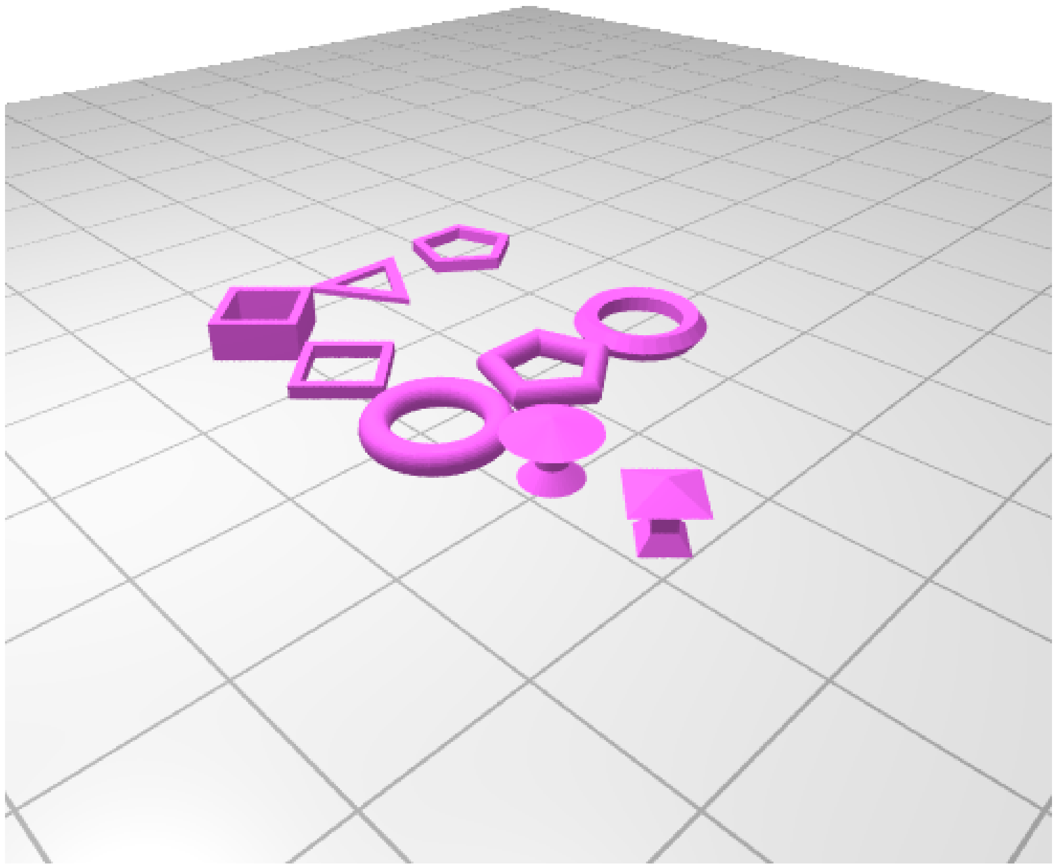
```
rectangular_extrude([ [10,10], [-10,10], [-20,0], [-10,-10], [10,-10] ],  // path is an a
rray of 2d coords
    {w: 1, h: 3, closed: true});
```

or more low-evel and object-oriented via rectangularExtrude(), with following unnamed variables: 1. width of the extrusion, in the z=0 plane 2. height of the extrusion in the z direction 3. resolution, number of segments per 360 degrees for the curve in a corner 4. roundEnds: if true, the ends of the polygon will be rounded, otherwise they will be flat

```
// first creating a 2D path, and then extrude it
var path = new CSG.Path2D([ [10,10], [-10,10], [-20,0], [-10,-10], [10,-10] ], /*closed=*
/true);
var csg = path.rectangularExtrude(3, 4, 16, true);   // w, h, resolution, roundEnds
return csg;
```

Rotate Extrude

Additional also rotate_extrude() is available:



```
// openscad-like
rotate_extrude( translate([4,0,0], circle({r: 1, fn: 30, center: true}) ) );

// more object-oriented
rotate_extrude({fn:4}, square({size: [1,1], center: true}).translate([4,0,0]) );

rotate_extrude( polygon({points:[ [0,0],[2,1],[1,2],[1,3],[3,4],[0,5] ]}) );
rotate_extrude({fn:4}, polygon({points:[ [0,0],[2,1],[1,2],[1,3],[3,4],[0,5] ]}) );
```

You essentially extrude any 2D polygon (circle, square or polygon).

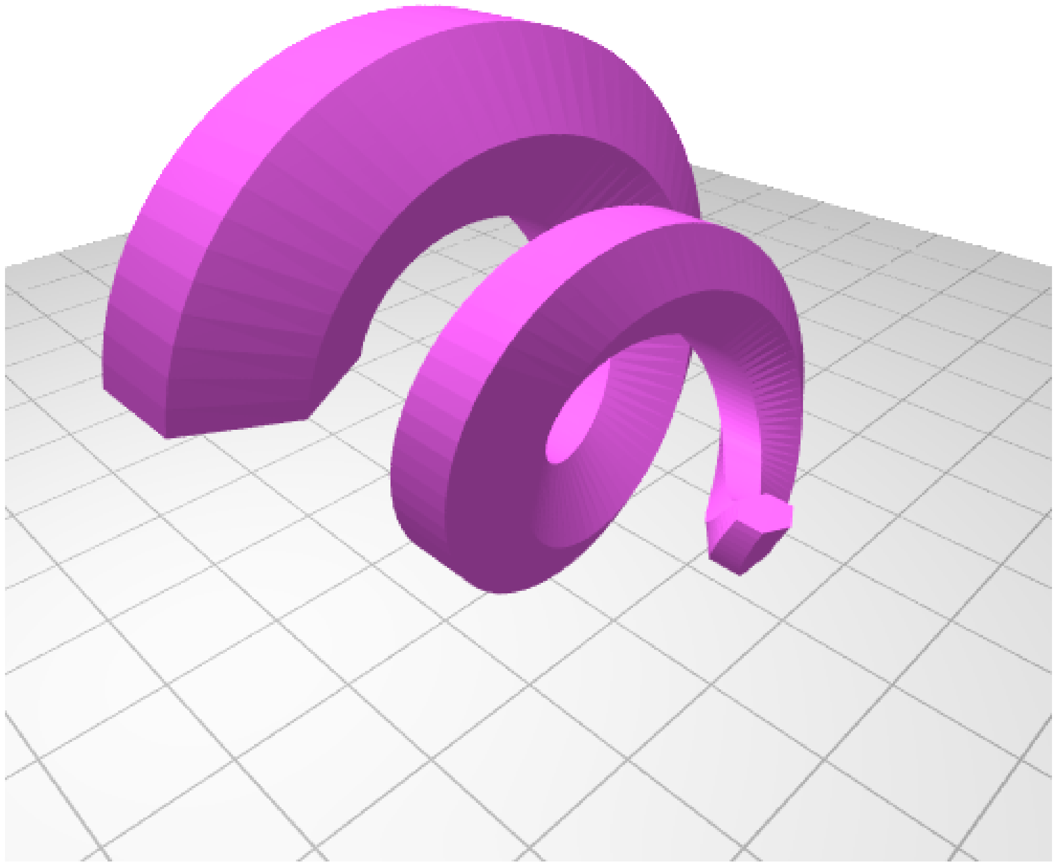
Solid from Slices

solidFromSlices() provides the ability to create a solid from a couple of polygons (e.g. also transition from a triangle to a square or any form), and those polygons can be transformed like with translate, and rotate while in transition.

solidFromSlices() is defined from CSG.Polygon instance and static method CSG.fromSlices, both methods generates a solid from provided options.

- @parameter {Object} options It provide slices to generate a solid. Each slice is a convex polygon in 3D space. The necessity of the convex limit is being evaluated. For proper result slices must be similar oriented - an angle between their normals should be less 180 degrees.
- numslices {Number} - number of slices to be generated. Must be 2 or more. The first and the last slices are used as bottom and top facets.
- callback {Function} - A function(t, sliceN) generating slices. It receives 2 parameters: t - [0..1], where 0 - corresponds the bottom facet and 1 - to the top. sliceN - [0..numslices-1] the current slice number. The callback is called in context of the CSG.Polygon. Thus slices can be generated from the initial polygon by transforming it. Look at "Tor (multi-color)" example.
- loop {Boolean} - If true the bottom and the top are not generated and a solid becomes a loop. Look at "Tor (multi-color)" example again.

```
var csg = new CSG.Polygon.createFromPoints([ [0,1,0], ... ]).  // initial polygon
    solidFromSlices({
      numslices: num,              // amount of slices
      loop: true|false,            // final CSG is close by looping (start = end) like a
torus
      callback: function(t,slice) {
         // t: 0..1
         // slice: 0 .. (numslices-1)
         return this.translate|rotate(...).setColor(..);
      }
   });
```



```
function main(params) {
    var sqrt3 = Math.sqrt(3) / 2;
    var radius = 10;

    var hex = CSG.Polygon.createFromPoints([
        [radius, 0, 0],
        [radius / 2, radius * sqrt3, 0],
        [-radius / 2, radius * sqrt3, 0],
        [-radius, 0, 0],
        [-radius / 2, -radius * sqrt3, 0],
        [radius / 2, -radius * sqrt3, 0]
    ]);

    var angle = 5;
    return hex.solidFromSlices({
      numslices: 720 / angle,
      callback: function(t, slice) {
         var coef = 1 - t * 0.8;
         return this.rotateZ(5 * slice).scale(coef).translate([radius * 4 * t, t * 15, 0]
).rotate(
             [0,20,0],
             [-1, 0, 0],
             angle * slice
         );
      }
   });
}
```

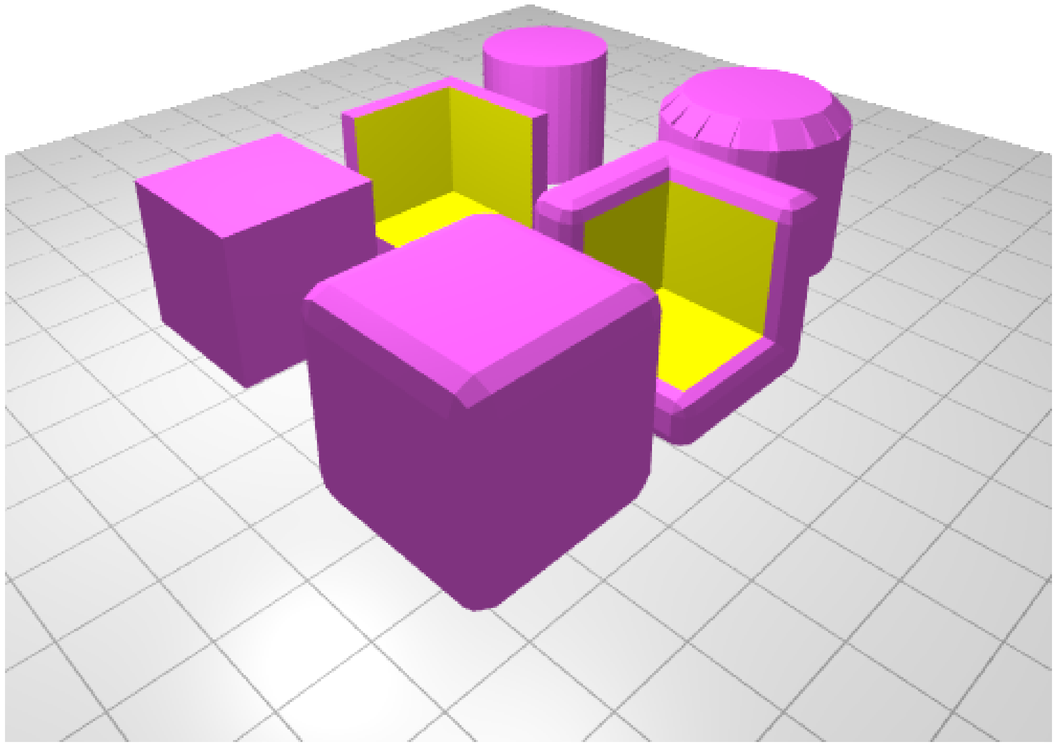
Expansion & Contraction

Expansion can be seen as the 3D convolution of an object with a sphere. Contraction is the reverse: the area outside the solid is expanded, and this is then subtracted from the solid.

Expansion and contraction are very powerful ways to get an object with nice smooth corners. For example a rounded cube can be created by expanding a normal cube.

Note that these are expensive operations: spheroids are created around every corner and edge in the original object, so the number of polygons quickly increases. Expansion and contraction therefore are only practical for simple non-curved objects.

expand() and contract() take two parameters: the first is the radius of expansion or contraction; the second parameter is optional and specifies the resolution (number of polygons on spherical surfaces, per 360 degree revolution):



```
expand(0.2, 8, difference(cube(2),translate([0.3,0.3,0.3], cube(2))));   // openscad like

var cube1 = CSG.cube({radius: 1.0});        // object-oriented
var cube2 = CSG.cube({radius: 1.0}).translate([-0.3, -0.3, -0.3]);
var csg = cube1.subtract(cube2);
var rounded = csg.expand(0.2, 8);
```

Properties

The 'property' property of a solid can be used to store metadata for the object, for example the coordinate of a specific point of interest of the solid. Whenever the object is transformed (i.e. rotated, scaled or translated), the properties are transformed with it. So the property will keep pointing to the same point of interest even after several transformations have been applied to the solid.

Properties can have any type, but only the properties of classes supporting a 'transform' method will actually be transformed. This includes CSG.Vector3D, CSG.Plane and CSG.Connector. In particular CSG.Connector properties (see below) can be very useful: these can be used to attach a solid to another solid at a predetermined location regardless of the current orientation.

It's even possible to include a CSG solid as a property of another solid. This could be used for example to define the cutout cylinders to create matching screw holes for an object. Those 'solid properties' get the same transformations as the owning solid but they will not be visible in the result of CSG operations such as union().

Other kind of properties (for example, strings) will still be included in the properties of the transformed solid, but the properties will not get any transformation when the owning solid is transformed.

All primitive solids have some predefined properties, such as the center point of a sphere (TODO: document).

The solid resulting from CSG operations (union(), subtract(), intersect()) will get the merged properties of both source solids. If identically named properties exist, only one of them will be kept.

```
var cube = CSG.cube({radius: 1.0});
cube.properties.aCorner = new CSG.Vector3D([1, 1, 1]);
cube = cube.translate([5, 0, 0]);
cube = cube.scale(2);
// cube.properties.aCorner will now point to [12, 2, 2],
// which is still the same corner point

// Properties can be stored in arrays; all properties in the array
// will be transformed if the solid is transformed:
cube.properties.otherCorners = [
  new CSG.Vector3D([-1, 1, 1]),
  new CSG.Vector3D([-1, -1, 1])
];

// and we can create sub-property objects; these must be of the
// CSG.Properties class. All sub properties will be transformed with
// the solid:
cube.properties.myProperties = new CSG.Properties();
cube.properties.myProperties.someProperty = new CSG.Vector3D([-1, -1, -1]);
```

Connectors

The CSG.Connector class is intended to facilitate attaching two solids to each other at a predetermined location and orientation. For example suppose we have a CSG solid depicting a servo motor and a solid of a servo arm: by defining a Connector property for each of them, we can easily attach the servo arm to the servo motor at the correct position (i.e. the motor shaft) and orientation (i.e. arm perpendicular to the shaft) even if we don't know their current position and orientation in 3D space.

In other words Connector give us the freedom to rotate and translate objects at will without the need to keep track of their positions and boundaries. And if a third party library exposes connectors for its solids, the user of the library does not have to know the actual dimensions or shapes, only the names of the connector properties.

A CSG.Connector consist of 3 properties:

- **point**: a CSG.Vector3D defining the connection point in 3D space
- **axis**: a CSG.Vector3D defining the direction vector of the connection (in the case of the servo motor example it would point in the direction of the shaft)
- **normal**: a CSG.Vector3D direction vector somewhat perpendicular to axis; this defines the "12 o'clock" orientation of the connection.

When connecting two connectors, the solid is transformed such that the **point** properties will be identical, the **axis** properties will have the same direction (or opposite direction if mirror == true), and the **normal**s match as much as possible.

Connectors can be connected by means of two methods: A CSG solid's **connectTo()** function transforms a solid such that two connectors become connected. Alternatively we can use a connector's **getTransformationTo()** method to obtain a transformation matrix which would connect the connectors. This can be used if we need to apply the same transform to multiple solids.

```
var cube1 = CSG.cube({radius: 10});
var cube2 = CSG.cube({radius: 4});

// define a connector on the center of one face of cube1
// The connector's axis points outwards and its normal points
// towards the positive z axis:
cube1.properties.myConnector = new CSG.Connector([10, 0, 0], [1, 0, 0], [0, 0, 1]);

// define a similar connector for cube 2:
cube2.properties.myConnector = new CSG.Connector([0, -4, 0], [0, -1, 0], [0, 0, 1]);

// do some random transformations on cube 1:
cube1 = cube1.rotateX(30);
cube1 = cube1.translate([3.1, 2, 0]);

// Now attach cube2 to cube 1:
cube2 = cube2.connectTo(
  cube2.properties.myConnector,
  cube1.properties.myConnector,
  true,   // mirror
  0       // normalrotation
);

// Or alternatively:
var matrix = cube2.properties.myConnector.getTransformationTo(
  cube1.properties.myConnector,
  true,   // mirror
  0       // normalrotation
);
cube2 = cube2.transform(matrix);

var result = cube2.union(cube1);
```

Bounds & Surface Laying

The getBounds() function can be used to retrieve the bounding box of an object. getBounds() returns an array with two CSG.Vector3Ds specifying the minimum x,y,z coordinate and the maximum x,y,z coordinate.

lieFlat() lays an object onto the z=0 surface, in such a way that the z-height is minimized and it is centered around the z axis. This can be useful for CNC milling: it will transform a part of an object into the space of the stock material during milling. Or for 3D printing: it is laid in such a way that it can be printed with minimal number of layers. Instead of lieFlat() the function getTransformationToFlatLying() can be used, which returns a CSG.Matrix4x4 for the transformation.

```
var cube1 = CSG.cube({radius: 10});
var cube2 = CSG.cube({radius: 5});

// get the right bound of cube1 and the left bound of cube2:
var deltax = cube1.getBounds()[1].x - cube2.getBounds()[0].x;

// align cube2 so it touches cube1:
cube2  = cube2.translate([deltax, 0, 0]);

var cube3 = CSG.cube({radius: [100,120,10]});
// do some random transformations:
cube3 = cube3.rotateZ(31).rotateX(50).translate([30,50,20]);
// now place onto the z=0 plane:
cube3  = cube3.lieFlat();

// or instead we could have used:
var transformation = cube3.getTransformationToFlatLying();
```

```
  cube3 = cube3.transform(transformation);

  return cube3;
```

Miscellaneous

Color

OpenSCAD-like:

- **color([r,g,b], object, object2 ...)** e.g. color([1,1,0],sphere());
- **color([r,g,b], array)**
- **color([r,g,b,a], object, object2 ...)**
- **color([r,g,b,a], array)**
- **color(name, object, object2 ...)** e.g. color('red',sphere());
- **color(name, a, object, object2 ...)** e.g. color('red',0.5, sphere());
- **color(name, array)**
- **color(name, a, array)**

whereas the named colors are case-insensitive ('RED'=='red')

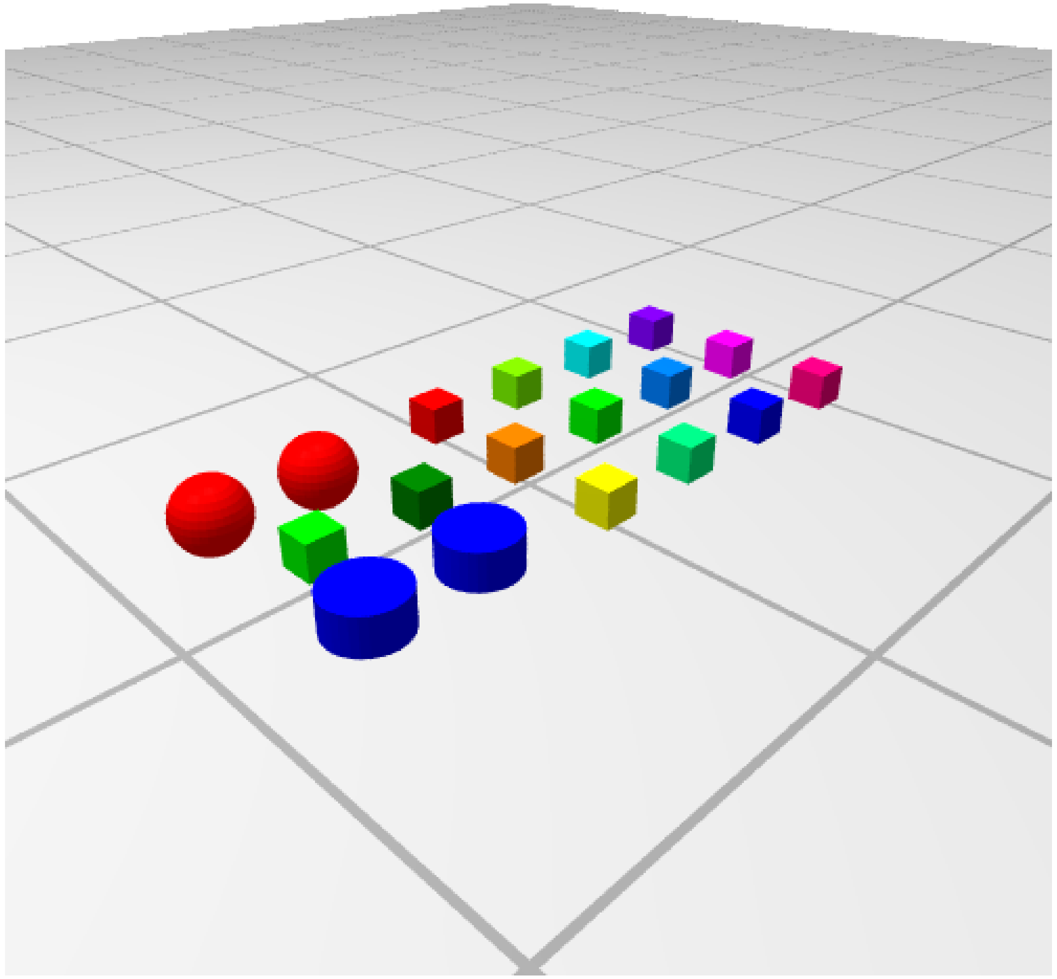
Object-oriented:

- **.setColor([r,g,b]);**
- **.setColor([r,g,b,a]);**
- **.setColor(r,g,b);**
- **.setColor(r,g,b,a);**

Examples:

```
color([1,0.5,0.3],sphere(1));                      // openscad like
color([1,0.5,0.3],sphere(1),cube(2));
color("Red",sphere(),cube().translate([2,0,0]));   // named color (case-insensitive)

sphere().setColor(1,0.5,0.3);                      // object-oriented
sphere().setColor([1,0.5,0.3]);
```

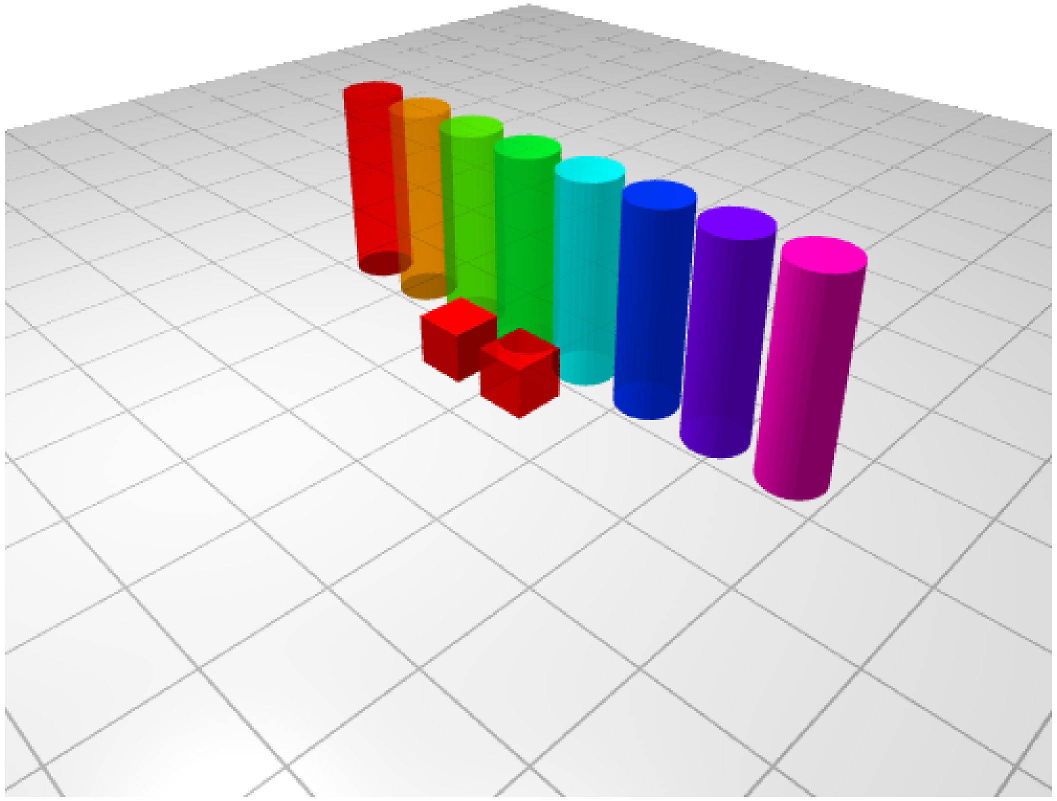
See Wikipedia: Web Colors for all available named colors sorted by tone.



code excerpt:

```
o.push( color([1,0,0],sphere()) );
o.push( color([0,1,0],cube()) );
o.push( color([0,0,1],cylinder()) );

o.push( color("red",sphere()) );
```

```
o.push( color("green", cube()) );
o.push( color("blue", cylinder()) );

for(var i=0; i<1; i+=1/12) {
   o.push( cube().setColor(hsl2rgb(i,1,0.5)) );
}
```



code:

```
function main() {
   var o = [];
   for(var i=0; i<8; i++) {
      o.push(cylinder({r:3,h:20}).
         setColor(
            hsl2rgb(i/8,1,0.5).  // hsl to rgb, creating rainbow [r,g,b]
            concat(1/8+i/8)      // and add to alpha to make it [r,g,b,a]
         ).translate([(i-3)*7.5,0,0])
      );
   }
   o.push(color("red",cube(5)).translate([-4,-10,0]));
   o.push(color("red",0.5,cube(5)).translate([4,-10,0]));
   return o;
}
```

Note: there are some OpenGL Transparency Limitations, e.g. depending on the order you might not see through otherwise partially transparent objects.

Color Spacing Conversion

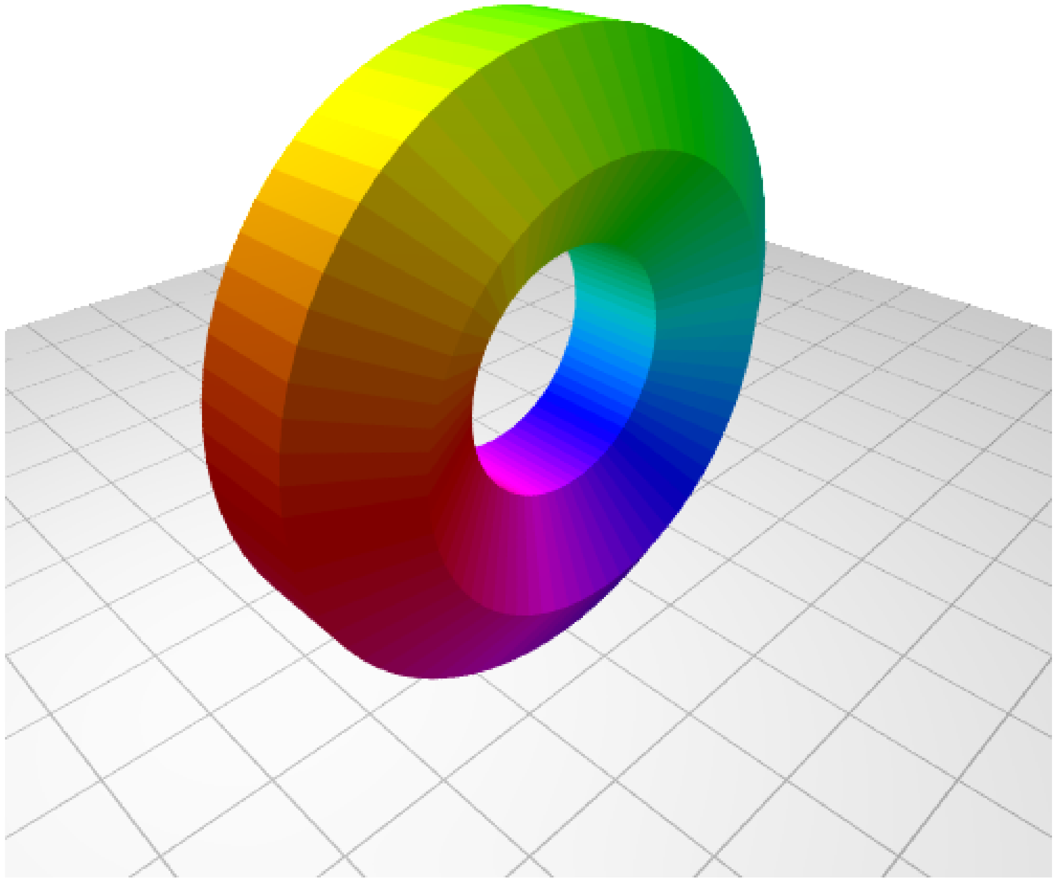
Following functions to convert between color spaces:

```
var hsl = rgb2hsl(r,g,b); // or rgb2hsl([r,g,b]);
var rgb = hsl2rgb(h,s,l); // or hsl2rgb([h,s,l]);
var hsv = rgb2hsv(r,g,b); // or rgb2hsv([r,g,b]);
var rgb = hsv2rgb(h,s,v); // or hsv2rgb([h,s,v]);
```

whereas

- r,g,b (red, green, blue)
- h,s,l (hue, saturation, lightness)
- h,s,v (hue, saturation, value)

E.g. to create a rainbow, t = 0..1 and .setColor(hsl2rgb(t,1,0.5))



See Tor (multi-color) example direct.

Echo

```
a = 1, b = 2;
echo("a="+a,"b="+b);
```

prints out on the JavaScript console:

```
a=1, b=2
```

Mathematical Functions

Following OpenSCAD compatible functions are available, aside of the Math.xyz() as of JavaScript:

```
sin(a);                    // a = 0..360
cos(a);                    //     ''
asin(a);                   // a = 0..1, returns 0..360
acos(a);                   //       ''
tan(a);                    // a = 0..360
atan(a);                   // a = 0..1, returns 0..360
atan2(a,b);                // returns 0..360
ceil(a);
floor(a);
abs(a);
min(a,b);
max(a,b);
rands(min,max,vn,seed);   // returns random vectors of vn dimension, seed not yet impleme
nted
log(a);
```

```
lookup(ix,v);             // ix = index, e.g. v = [ [0,100], [10,10], [20,200] ] whereas
v[x][0] = index, v[x][1] = value
                          //    return will be linear interpolated (e.g. lookup(5,[ [0,10
0], [10,10], [20,200] ]) == 45

pow(a,b);
sign(a);                  // -1, 0 or 1
sqrt(a);
round(a);
```

Direct OpenSCAD Source Import

An OpenSCAD (.scad) translator & importer is included in OpenJSCAD, following features aren't working yet:

- DXF import and manipulation (e.g. import_dxf, dxf-cross, dxf_dim functions).
- rotate_extrude() (**Note:** OpenJSCAD supports rotate_extrude())
- minkowski() and hull() transformations (**Note:** OpenJSCAD supports hull())
- $fa, $fs global variables.
- Modifier characters: #, !, %

You can edit OpenSCAD source in the built-in editor, just make sure the first line says:

```
//!OpenSCAD
```

then the source-code is considered OpenSCAD syntax.

Further CAD languages support might arrive at a later time.

Converting OpenSCAD to OpenJSCAD

In order to translate your OpenSCAD .scad into native OpenJSCAD .jscad code, consider this comparison:

OpenSCAD (.scad)

```
union() {
      //cube(size=[30,30,0.1],center=true);
      translate([3,0,0]) cube();
      difference() {
         rotate([0,-45,0]) cube(size=[8,7,3],center=true);
         sphere(r=3,$fn=20,center=true);
      }
      translate([10,5,5]) scale([0.5,1,2]) sphere(r=5,$fn=50);
      translate([-15,0,0]) cylinder(r1=2,r2=0,h=10,$fn=20);

   for(i=[0:19]) {
      rotate([0,i/20*360,0])
      translate([i,0,0])
      rotate([0,i/20*90,i/20*90,0])
      cube(size=[1,1.2,.5],center=true);
   }
}
```

OpenJSCAD (.jscad)

```
function main() {
   var cubes = new A
   for(i=0; i<20; i+
      cubes[i] = rot
         translate([
         rotate([0,i
         cube({size:
   }
   return union(
      //cube({size:[
      translate([3,0
      difference(
         rotate([0,-
         sphere({r:3
      ),
      translate([10,
      translate([-15
      cubes
   );
}
```

Essentially whenever named arguments in .scad appear func(a=1), translate it into func({a:1}), for example:

- .scad: `translate([0,0,2]) sphere(size=2,$fn=50);`
- .jscad (1): `translate([0,0,2], sphere({size:2,fn:50}));`
- .jscad (2): `sphere({size:2,fn:50}).translate([0,0,2]);`

Interactive Parametric Models

Models can have interactive parameters, allowing users to change values via familiar forms, i.e. typing in numbers, sliding bars, pulling down menus, etc. This allows the user to change values and create any number of possible combinations, allowing the model to become dynamic in nature. Any number of custom designs can be created, which can then be down loaded in any supported format.

Interactive parameters are possible by adding a specific function called getParameterDefinitions(). This function can be added anywhere in to your JSCAD script, but must return an array of parameter definitions.

```
function getParameterDefinitions() {
  return [{ name: 'width', type: 'float', initial: 10, caption: "Width of the cube:" }];
}
```

The parameter definitions are used to create a set of fields which the user can change, i.e. options. The values of the fields are supplied to the main() function of your JSCAD script. For example, the value of the 'width' parameter is supplied as the 'params.width' attribute, and so on.

```
function main(params) {
    // custom error checking:
    if(params.width <= 0) throw new Error("Width should be positive!");
    var mycube = CSG.cube({radius: params.width});
    return mycube();
  }
```

All the common HTML5 field types are available as interactive parameters. This includes checkbox, color, date, email, float, int, number, password, slider, text and url. As well as two special parameter types for pull down choices, and grouping parameters.

A minimum parameter specification contains only 'name' and 'type'. However, a complete parameter specification should have a 'caption' and 'initial' value. In addition, there are 'min', 'max', 'step', 'checked', 'size, 'maxlength', and 'placeholder' which are relevant to specific parameter types. Just keep try different combinations to get a nice parameter specification. Here's an good example.

```
  {
    name: 'width',
    type: 'float',
    initial: 1.20,
    caption: 'Width of the thingy:',
    min: 1.0,
    max: 5.0,
    step: 0.5
  }
```

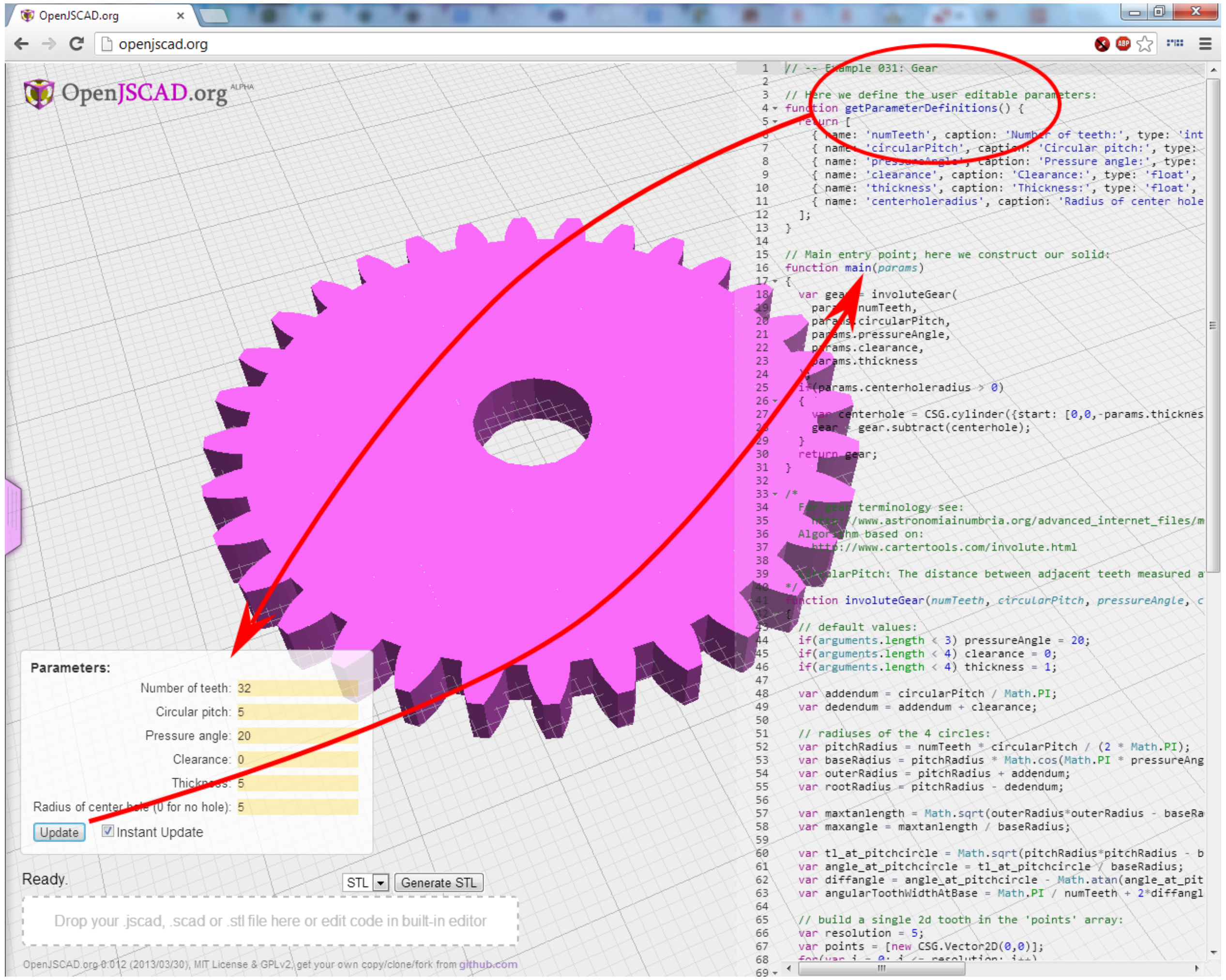
In addition, there is the 'choice' type which creates a drop down list of choices for the user. The choices are provided by specifying the 'values' and 'captions'. The chosen value is passed into the main() function of the model.

```
{
  name: 'shape',
  type: 'choice',
  values: ["TRI", "SQU", "CIR"],              // these are the values that will be suppl
ied to your script
  captions: ["Triangle", "Square", "Circle"],  // optional, these values are shown in the
 listbox
                                              // if omitted, the items in the 'values' a
rray are used
  caption: 'Shape:',                          // optional, displayed left of the input f
ield
  initial: "SQU",                             // optional, default selected value
                                              // if omitted, the first item is selected
by default
                                              // NOTE: parameter "default" is deprecated
}
```

A complete example:

```
function getParameterDefinitions() {
  return [
    { name: 'width', type: 'float', initial: 10, caption: "Width of the cube:" },
    { name: 'height', type: 'float', initial: 14, caption: "Height of the cube:" },
    { name: 'depth', type: 'float', initial: 7, caption: "Depth of the cube:" },
    { name: 'rounded', type: 'choice', caption: 'Round the corners?', values: [0, 1], cap
tions: ["No thanks", "Yes please"], initial: 1 }
  ];
}

function main(params) {
  var result;
  if(params.rounded == 1) {
    result = CSG.roundedCube({radius: [params.width, params.height, params.depth], roundr
adius: 2, resolution: 32});
  } else {
    result = CSG.cube({radius: [params.width, params.height, params.depth]});
  }
  return result;
}
```

There are plenty of examples of "Interactive" parameters available at http://www.openjscad.org/



```
// -- Example 031: Gear

// Here we define the user editable parameters:
function getParameterDefinitions() {
  return [
    { name: 'numTeeth', caption: 'Number of teeth:', type: 'int
    { name: 'circularPitch', caption: 'Circular pitch:', type:
    { name: 'pressureAngle', caption: 'Pressure angle:', type:
    { name: 'clearance', caption: 'Clearance:', type: 'float',
    { name: 'thickness', caption: 'Thickness:', type: 'float',
    { name: 'centerholeradius', caption: 'Radius of center hole
  ];
}

// Main entry point; here we construct our solid:
function main(params)
{
  var gear = involuteGear(
    params.numTeeth,
    params.circularPitch,
    params.pressureAngle,
    params.clearance,
    params.thickness
  );
  if(params.centerholeradius > 0)
  {
    var centerhole = CSG.cylinder({start: [0,0,-params.thicknes
    gear = gear.subtract(centerhole);
  }
  return gear;
}

/*
  For gear terminology see:
    http://www.astronomiainumbria.org/advanced_internet_files/m
  Algorithm based on:
    http://www.cartertools.com/involute.html

  circularPitch: The distance between adjacent teeth measured a
*/
function involuteGear(numTeeth, circularPitch, pressureAngle, c
{
  // default values:
  if(arguments.length < 3) pressureAngle = 20;
  if(arguments.length < 4) clearance = 0;
  if(arguments.length < 4) thickness = 1;

  var addendum = circularPitch / Math.PI;
  var dedendum = addendum + clearance;

  // radiuses of the 4 circles:
  var pitchRadius = numTeeth * circularPitch / (2 * Math.PI);
  var baseRadius = pitchRadius * Math.cos(Math.PI * pressureAng
  var outerRadius = pitchRadius + addendum;
  var rootRadius = pitchRadius - dedendum;

  var maxtanlength = Math.sqrt(outerRadius*outerRadius - baseRa
  var maxangle = maxtanlength / baseRadius;

  var tl_at_pitchcircle = Math.sqrt(pitchRadius*pitchRadius - b
  var angle_at_pitchcircle = tl_at_pitchcircle / baseRadius;
  var diffangle = angle_at_pitchcircle - Math.atan(angle_at_pit
  var angularToothWidthAtBase = Math.PI / numTeeth + 2*diffangl

  // build a single 2d tooth in the 'points' array:
  var resolution = 5;
  var points = [new CSG.Vector2D(0,0)];
  for(var i = 0; i <= resolution; i++)
```

Parameters:

Number of teeth:	32
Circular pitch:	5
Pressure angle:	20
Clearance:	0
Thickness:	5
Radius of center hole (0 for no hole):	5

Update [✓ Instant Update]

Ready. STL Generate STL

Drop your .jscad, .scad or .stl file here or edit code in built-in editor

OpenJSCAD.org 0.012 (2013/03/30), MIT License & GPLv2, get your own copy/clone/fork from github.com

Command Line Interface (CLI) & Parameters

`openjscad` accepts parameter for the interactive parameters:

- --*key value* (e.g. **--thickness 3**) or
- --*key=value* (e.g. **--thickness=3**)

For example:

```
% openjscad name_plate.jscad --name "Just Me" --title "Geek" -o JustMe.amf
% openjscad name_plate.jscad "--name=Just Me" "--title=Geek" -o JustMe.amf
```

Orthonormal Basis

An orthonormal basis can be used to convert 3D points to 2D points by projecting them onto a 3D plane. An orthonormal basis is constructed from a given plane. Optionally a 'right hand' vector can be given, this will become the x axis of the two dimensional plane. If no right hand vector is given, a random one is chosen.

CSG.OrthoNormalBasis.Z0Plane() creates an orthonormal basis for the z=0 plane. This transforms (xx,yy,zz) 3D coordinates into the 2D (xx, yy) coordinates, or vice versa from (xx, yy) into (xx, yy, 0).

Use to2D() and line3Dto2D() to convert from the 3D space to the 2D plane. Use to3D() and line2Dto3D() to convert the other way.

getProjectionMatrix() gives the projection matrix to transform into the orthonormal basis. getInverseProjectionMatrix() gives the matrix to transform back into the original basis.

```
// construct a plane:
var plane = CSG.Plane.fromNormalAndPoint([1,1,0], [0,0,1]);
var orthobasis = new CSG.OrthoNormalBasis(plane);
// or if we would like a specific right hand vector:
// var orthobasis = new CSG.OrthoNormalBasis(plane, [0,0,1]);
```

```
var point3d = new CSG.Vector3D(1,5,7);
var point2d = orthobasis.to2D(point3d);
var projected = orthobasis.to3D(point2d);
```

2D & 3D Math

There are utility classes for many 2D and 3D operations. Below is a quick summary, for details view the source of csg.js:

Vector3D

```
var vec1 = new CSG.Vector3D(1,2,3);       // 3 arguments
var vec2 = new CSG.Vector3D( [1,2,3] );   // 1 array argument
var vec3 = new CSG.Vector3D(vec2);        // cloning a vector
// get the values as: vec1.x, vec.y, vec1.z
// vector math. All operations return a new vector, the original is unmodified!
// vectors cannot be modified. Instead you should create a new vector.
vec.negated()
vec.abs()
vec.plus(othervector)
vec.minus(othervector)
vec.times(3.0)
vec.dividedBy(-5)
vec.dot(othervector)
vec.lerp(othervector, t)  // linear interpolation (0 <= t <= 1)
vec.length()
vec.lengthSquared()       // == vec.length()^2
vec.unit()
vec.cross(othervector)    // cross product: returns a vector perpendicular to both
vec.distanceTo(othervector)
vec.distanceToSquared(othervector)  // == vec.distanceTo(othervector)^2
vec.equals(othervector)
vec.multiply4x4(matrix4x4)   // right multiply by a 4x4 matrix
vec.min(othervector)        // returns a new vector with the minimum x,y and z values
vec.max(othervector)        // returns a new vector with the maximum x,y and z values
```

Vector2D

```
var vec1 = new CSG.Vector2D(1,2);       // 2 arguments
var vec2 = new CSG.Vector2D( [1,2] );   // 1 array argument
var vec3 = new CSG.Vector2D(vec2);      // cloning a vector
// vector math. All operations return a new vector, the original is unmodified!
vec.negated()
vec.abs()
vec.plus(othervector)
vec.minus(othervector)
vec.times(3.0)
vec.dividedBy(-5)
vec.dot(othervector)
vec.lerp(othervector, t)  // linear interpolation (0 <= t <= 1)
vec.length()
vec.lengthSquared()       // == vec.length()^2
vec.unit()
vec.normal()              // returns a 90 degree clockwise rotated vector
vec.distanceTo(othervector)
vec.distanceToSquared(othervector)  // == vec.distanceTo(othervector)^2
vec.cross(othervector)    // 2D cross product: returns a scalar
vec.equals(othervector)
vec.min(othervector)        // returns a new vector with the minimum x and y values
vec.max(othervector)        // returns a new vector with the maximum x and y values
vec.multiply4x4(matrix4x4)   // right multiply by a 4x4 matrix
vec.toVector3D(z)         // convert to a vector3D by adding a z coordinate
vec.angleDegrees()        // returns the angle of the vector: [1,0] = 0 degrees, [0, 1] =
 90 degrees, etc
vec.angleRadians()        // ditto in radians
var vec = CSG.Vector2D.fromAngleDegrees(degrees);  // returns a vector at the specified a
ngle
var vec = CSG.Vector2D.fromAngleRadians(radians);  // returns a vector at the specified a
ngle
```

Matrix4x4

```
var m1 = new CSG.Matrix4x4();          // unity matrix
var m2 = new CSG.Matrix4x4( [1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1] );
  // elements are passed in row order
var result = m1.plus(m2);
var result = m1.minus(m2);
var result = m1.multiply(m2);
// matrix vector multiplication (vectors are padded with zeroes to get a 4x1 vector):
var vec3d = m1.rightMultiply1x3Vector(vec3d);  // matrix * vector
var vec3d = m1.leftMultiply1x3Vector(vec3d);   // vector * matrix
var vec2d = m1.rightMultiply1x2Vector(vec2d);  // matrix * vector
var vec2d = m1.leftMultiply1x2Vector(vec2d);   // vector * matrix
// common transformation matrices:
var m = CSG.Matrix4x4.rotationX(degrees);      // matrix for rotation about X axis
var m = CSG.Matrix4x4.rotationY(degrees);      // matrix for rotation about Y axis
var m = CSG.Matrix4x4.rotationZ(degrees);      // matrix for rotation about Z axis
var m = CSG.Matrix4x4.rotation(rotationCenter, rotationAxis, degrees); // rotation about
arbitrary point and axis
var m = CSG.Matrix4x4.translation(vec3d);      // translation
var m = CSG.Matrix4x4.scaling(vec3d);          // scale
var m = CSG.Matrix4x4.mirroring(plane);        // mirroring in a plane; the argument must
 be a CSG.Plane
// matrix transformations can be concatenated:
```

```
var transform = CSG.Matrix4x4.rotationX(20).multiply(CSG.Matrix4x4.rotationY(30));
// Use a CSG solid's transform() method to apply the transformation to a CSG solid
```

Plane

A 3D plane is represented by a normal vector (should have unit length) and a distance from the origin w, the plane passes through normal.times(w)

```
var plane1 = new CSG.Plane(normal, w);
// Or we can construct a plane from 3 points:
var plane2 = CSG.Plane.fromPoints(p1, p2, p3);
// Or from a normal vector and 1 point:
var plane3 = CSG.Plane.fromNormalAndPoint(normal, point);
// Flip a plane (front side becomes back side):
var plane4 = plane3.flipped();
// Apply transformations (rotation, scaling, translation):
var transformed = plane3.transformed(matrix4x4);  // argument is a CSG.Matrix4x4
// Intersection of plane and 3d line:
var point = plane3.intersectWithLine(line);        // argument is CSG.Line3D, returns a C
SG.Vector3D
// Intersection of 2 planes:
var line = plane3.intersectWithPlane(plane);       // argument is another CSG.Plane, retu
rns a CSG.Line3D
// Distance to point:
var w = signedDistanceToPoint(point);             // argument is CSG.Vector3D, returns a
float (positive
                                                  //    if in front of plane, negative if
 in back)
```

Line3D

A line in 3d space is represented by a point and a direction vector. Direction should be a unit vector. Point can be any point on the line:

```
var line = new CSG.Line3D(point, direction);      // argumenst are CSG.Vector3D
// or by giving two points:
var line = CSG.Line3D.fromPoints(p1, p2);         // argumenst are CSG.Vector3D
var point = intersectWithPlane(plane);            // == plane.intersectWithLine(this)
var line2 = line.reverse();                       // same line but reverse direction
var line2 = line.transform(matrix4x4);            // for rotation, scaling, etc
var p = line.closestPointOnLine(point);           // project point onto the line
var d = line.distanceToPoint(point);
```

Line2D

A line in 2d space is represented by a normal vector and a distance w to the origin along the normal vector, or by a point and a direction vector. Direction should be a unit vector. Point can be any point on the line:

```
var line = new CSG.Line2D(CSG.Line2D(normal,w));
// or by giving two points:
var line = CSG.Line2D.fromPoints(p1, p2);         // argumenst are CSG.Vector2D
var line2 = line.reverse();                       // same line but reverse direction
var line2 = line.transform(matrix4x4);            // for rotation, scaling, etc
var point = line.origin();                        // returns the point closest to the ori
gin
var dir = line.direction();                       // direction vector (CSG.Vector2D)
var x = line.xAtY(y);                             // returns the x coordinate of the line
 at given y coordinate
var d = absDistanceToPoint(point);                // returns the absolute distance betwee
```

```
n a point and the line
var p = line.closestPoint(point);                 // projection of point onto the line
var point = line.intersectWithLine(line2);        // intersection of two lines, returns C
SG.Vector2D
```

Including Files

include() allows to include other JSCAD files (also recursively), e.g.

```
// main.jscad

include("lib.jscad");

function main() {
   return myLib.b(2);
}
```

and

```
// lib.jscad

myLib = function() {
   var a = function(n) {  // internal
      return n*2;
   }
   myLib.b = function(n) {      // public
      return sphere(a(n));
   }
}
```

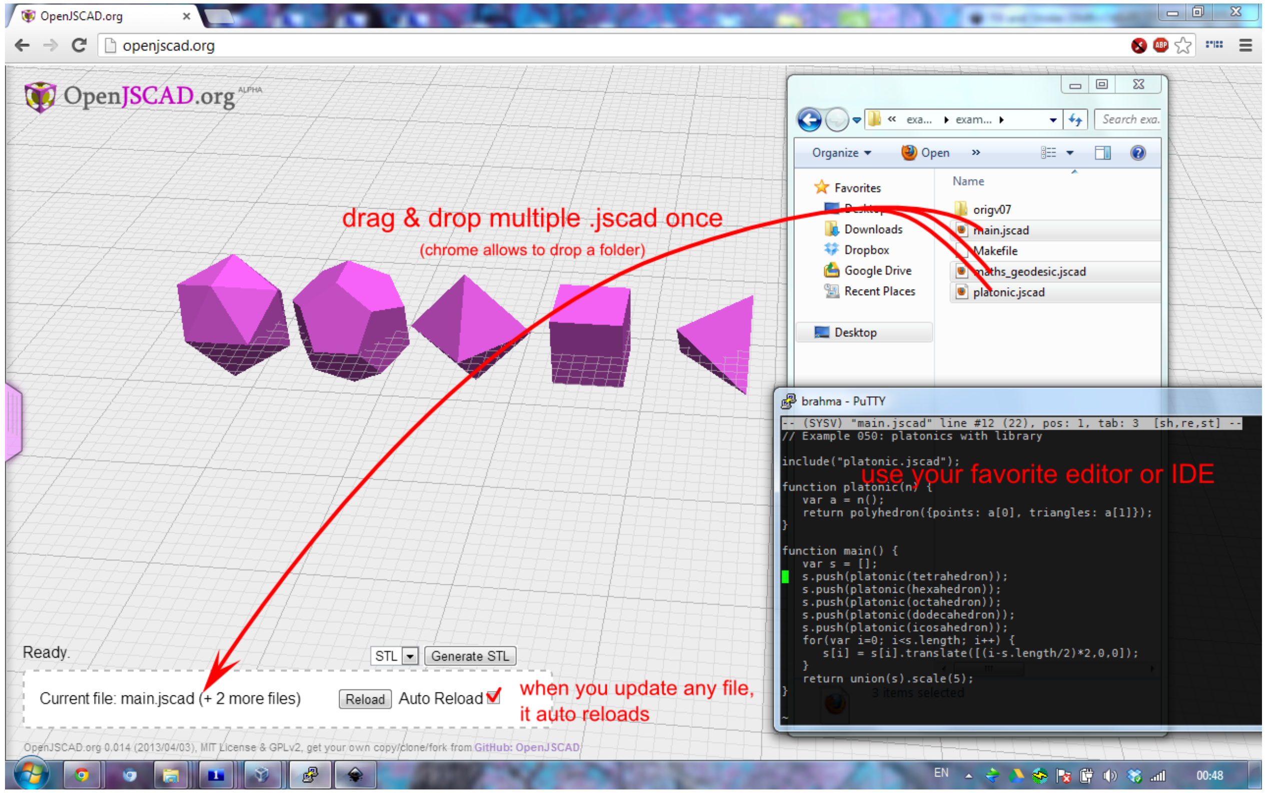
See also Example 50: Platonics with a recursive use of include(); yet, it's a rather bad example of not localize the function names. A clear writing style-guide will follow how an OpenJSCAD library should look like.

Support of include()

include() is supported

- **web-online remote** (e.g. http://openjscad.org/): given you drag & dropped the files, or they are available on the web-server (e.g. the examples)
- **web-online local** (e.g. http://localhost/OpenJSCAD/): given you drag & dropped the files, or they are available on the local web-server
- **web-offline local** (e.g. file://..../OpenJSCAD/index.html): given you drag & dropped the files
- **command-line interface** (CLI): given they are locally available in the filesystem

Example of a setup with drag & dropped files:


OpenJSCAD.org
openjscad.org
OpenJSCAD.org ALPHA
drag & drop multiple .jscad once
(chrome allows to drop a folder)
Organize
Open
Favorites
Downloads
Dropbox
Google Drive
Recent Places
Desktop
Name
origv07
main.jscad
Makefile
maths_geodesic.jscad
platonic.jscad
brahma - PuTTY
use your favorite editor or IDE
Ready.
STL
Generate STL
Current file: main.jscad (+ 2 more files)
Reload
Auto Reload
when you update any file,
it auto reloads
OpenJSCAD.org 0.014 (2013/04/03), MIT License & GPLv2, get your own copy/clone/fork from GitHub: OpenJSCAD

```
-- (SYSV) "main.jscad" line #12 (22), pos: 1, tab: 3  [sh,re,st] --
// Example 050: platonics with library

include("platonic.jscad");

function platonic(n) {
   var a = n();
   return polyhedron({points: a[0], triangles: a[1]});
}

function main() {
   var s = [];
   s.push(platonic(tetrahedron));
   s.push(platonic(hexahedron));
   s.push(platonic(octahedron));
   s.push(platonic(dodecahedron));
   s.push(platonic(icosahedron));
   for(var i=0; i<s.length; i++) {
      s[i] = s[i].translate([(i-s.length/2)*2,0,0]);
   }
   return union(s).scale(5);
}
```

File Layout of JSCAD Files

Assuming you want to create a larger OpenJSCAD project, you might use **include()** to split up the functionality:

```
ProjectName/
   main.jscad          # this one contains the "function main()", this file will be execu
ted
   addon.jscad         # this file could be include("addon.jscad") in main.jscad
   optimizer.jscad     #             ''     include("optimizer.jscad") in main.jscad or a
lso in addon.jscad
```

```
     Makefile            # possible Makefile to do the same on CLI
```

Note: main.jscad will be the default file which will be executed, and has to contain "function main()" declaration.

Developing with Multiple JSCAD Files

Depending on your browser and your local setup, following applies:

- **Chrome (Version 26+)**:
    - Online (http://...): drag & drop entire folder, e.g. ProjectName/ to the drag & drop zone
    - Offline (file://...): drag & drop all jscad files (but not folder) to the drag & drop zone
- **Firefox (Version 19+)**: drag & drop all jscad files of the project to the drag & drop zone
- **Opera**: not yet working (WebGL support not yet available)
- **IE10**: not yet working (WebGL support not yet available)

Addendum

- OpenJsCad Documentation, object-oriented only approach, most information from there is included here too

-- End of User Guide --